Dynamic Object Flow Analysis

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Adrian Lienhard

von Biel/Bözingen (BE)

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 16.12.2008

Der Dekan:
Prof. Dr. U. Feller

This dissertation is available as a free download from http://scg.unibe.ch/

ISBN 978-1-4092-4742-5
First Edition, December 2008

Acknowledgments

I am grateful to Oscar Nierstrasz for giving me the opportunity to work at the Software Composition Group. Oscar, thanks for your excellent support and rigorous feedback on my work.

I may not have taken the path of pursuing a PhD without Stéphane Ducasse, who introduced me to Smalltalk and the Software Composition Group. Thanks, Stef, for all your encouragement and enthusiasm.

I am grateful to Wim De Pauw for being the external reviewer of this thesis and for coming to Switzerland to join the jury of the PhD defense. Also, I thank Matthias Zwicker for accepting to chair the examination.

I thank Tudor Gîrba for all our inspiring discussions and for providing many creative ideas that have influenced this work.

I am much obliged to the following people that provided appreciated feedback on drafts of this dissertation: Orla Greevy, Tudor Gîrba, Daniel Ratiu, and Adrian Kuhn.

Many thanks go to the present and former Software Composition Group members. Marcus Denker, Tudor Gîrba, Adrian Kuhn, Fabrizio Perin, Lukas Renggli, Jorge Ressia, David Röthlisberger, Toon Verwaest, Gabriela Arévalo, Alexandre Bergel, Markus Gälli, Orla Greevy, Michele Lanza, Laura Ponisio, and Nathanael Schärli. We shared many interesting discussions, relaxing coffee times, and nice barbecues. It is a great group!

Thanks also to Therese Schmid and Iris Keller for their excellent support with the administrative chores.

I had nice moments and discussions with people I met at conferences. Thanks to Daniel Ratiu, Mircea Trifu, Adrian Dozsa for introducing me to the Rumanian connection of Athens, and thanks to Marco D'Ambros, Michele Lanza, Daniel Ratiu, Romain Robbes, Richard Wettel, for the bear hunting trip in Canada.

Special thanks go to Christoph Wysseier and all netstyle.ch and Cmsbox co-workers. They relieved me of a lot of work so I could focus on my PhD.

I am deeply grateful to my parents and my sister who have unconditionally supported me over all those years.

Above all, I thank my love, Felicia Flicker, for being in my life and for sharing many unforgettable moments.

Adrian Lienhard
November 17, 2008

Abstract

The behavior of an object-oriented software system is notoriously hard to understand from the source code alone. The main reason is the large gap between the program's static structure and its actual runtime behavior. Features inherent to object-orientation, like object aliasing and late binding, — while providing a high degree of expressiveness to model an application domain — make programs hard to understand, maintain, and analyze.

Complementary to static analysis, dynamic analysis can help to close this gap by investigating the properties of a running program. The state of the art in dynamic analysis focuses on investigating runtime control flow and structures of object graphs, but a thorough analysis of how objects are passed through a system is missing. Tracking how object references are transferred, however, is essential to analyze the dependencies introduced by object aliasing.

In this dissertation we propose Object Flow Analysis, our approach to track object flow by explicitly representing object references and reference transfer. Object Flow Analysis provides an effective and original way of analyzing and runtime monitoring dependencies introduced by object aliasing. To validate Object Flow Analysis, we propose three different reverse engineering applications that, based on Object Flow Analysis, reason about aliasing dependencies in object-oriented programs. Yet Object Flow Analysis extends beyond traditional reverse engineering applications. A key contribution of our work is that we advance the state of the art in back-in-time debugging by proposing and providing an implementation of the concept of Object Flow Analysis in a high-level language virtual machine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

I do not believe in things. I believe only in their relationships.
— George Braque

Dynamic software analysis is commonly referred to as the analysis of the properties of a running program [BALL 99]. In contrast, static analysis is the process of analyzing source code to predict safe approximations to its behavior during execution.

The Price *et al.* taxonomy [PRIC 93] for program visualization distinguishes four dimensions to categorize program behavior: (1) control instructions, (2) flow of control, (3) data structures, and (4) flow of data. Figure 1.1 illustrates how static and dynamic analyses cover these dimensions. While static analysis completely covers both control and data dimensions (intra- and inter-procedural control flow analysis; points-to analysis), dynamic analysis approaches exist for the first three categories — but there is a gap in the data flow category.

Dynamic control analysis records the statements executed at runtime. Examples are the detection of the code coverage of tests [REIC 07] and dynamic slicing [KORE 97]. Control flow analysis — also referred to as method execution tracing — is widely used for program comprehension and reverse engineering [HAMO 04]. Dynamic data analyses mainly investigate the structure of memory snapshots to detect memory leaks or to analyze encapsulation properties [DE P 00, HILL 02]. There also exist approaches that cover both control flow and data dimensions, for example to implement back-in-time debuggers [LEWI 03], which allow the user to go back in the

	control	control flow	data	data flow
static analyis	✔	✔	✔	✔
dynamic analyis	✔	✔	✔	?

Figure 1.1: The analysis of control and data flow in static and dynamic analysis. Dynamic analysis approaches lack information about object flow.

history of a program execution. Although in recent years there has been an increasing research effort dedicated to the dynamic analysis of object-oriented software, dynamic data flow analysis has attracted little attention so far.

1.1 The Problem of Object Aliasing

We have focused our research on the dynamic analysis of *object-oriented* software systems because understanding object-oriented systems comes with additional challenges [WILD 92] compared to non object-oriented systems.

The power of the object-oriented programming language paradigm lies in the flexibility of the interconnecting structure of objects. A typical object-oriented program creates a large number of objects and an even larger number of references between them. The memory structure formed by the objects on the heap, commonly referred to as *object graph*, can be a complicated structure and its topology evolves over time. New objects are created and join the object graph and old objects leave the graph. As the techniques of object-oriented programming have matured, the design of structures of interacting objects has come to be seen as at least as important as the design of the objects themselves and their classifications using inheritance [HILL 02].

The situation that occurs when two or more objects reference the same object is called *object aliasing* [HOGG 92]. Object aliasing provides a high degree of flexibility to model an application domain because it enables sharing of mutable state. But this flexibility comes at a cost. Because an object can be accessed and modified from any object that holds a reference to it, and references are transferred at runtime, object-oriented programs can be hard to understand, maintain, and analyze.

Object aliasing introduces *dependencies* between software entities (*e.g.*, between classes). A dependency between entities X and Y means that a developer modifying the implementation of X must be concerned about possible side effects in Y [WILD 92].

Dependencies caused by object aliasing are difficult to detect from reading the source code because of the large gap between a program's static representation and its actual runtime behavior. This is a particular problem in the context of legacy systems in which the maintainer can only form a local understanding of the program, focusing his attention on the portion of the code in which the repair or enhancement is to be made. By overlooking dependencies, he may introduce subtle bugs in seemingly unrelated parts of the program [LETO 86]. Therefore, pervasive object aliasing remains a major source of software defects [GROT 01] and complicates program comprehension [DOLA 03].

The following two kinds of dependencies between objects, which occur when object encapsulation is broken, are well known:

- Representation exposure — an object's representation is exposed outside the scope of its implementation [LISK 86].
- External dependency — external objects are part of an object's invariant [NOBL 98].

In both cases, modifying the object depended on through a reference that resides outside the encapsulation boundary may violate an assumption of the implementation and hence break the program.

Object encapsulation is generally considered to prevent these problems but in today's object-oriented mainstream languages, like Java and C#, there exists no mechanism to guarantee that objects do not inadvertently escape their encapsulation scope. Motivated by this problem, type systems that allow one to statically control ownership have been an active area of research during the last 20 years [HOGG 91, NOBL 98, CLAR 01, BOYA 03].

Faced with new ownership type systems and large legacy applications, techniques have been sought to automatically infer ownership annotations for existing code bases [ALDR 02]. Statically analyzing ownership properties of existing code has turned out to be a hard problem in practice because of the limited capabilities of static analysis to precisely predict the actual behavior of a program [CLAR 07]. Dynamic analysis examines exactly this behavior. More recently, therefore, research has been invested into the *dynamic analysis* of the runtime structure of objects — not only into the dynamic inference of ownership annotations [AGAR 04, DIET 07], but also into more general analyses of object reference structures [DE P 00, HILL 02, MITC 06, PHEN 06]. With these approaches, the encapsulation structure and aliasing dependencies in object graph snapshots can be analyzed.

However, aliasing dependencies exist that can be difficult to detect by analyzing only the reference structure of object graphs. If aliasing dependencies between software entities exist that are not only spatially but also temporally disconnected, object graph snapshots provide only part of the necessary data because they lack the notion of time. We can identify such aliasing dependencies in various program abstractions:

- *Static program elements* (methods, classes, packages, etc.) — these elements can depend on each other by transferring objects.
- *Software features* — a feature may depend on the object references produced by a previously exercised feature.
- *Control flow* — the execution of a method may influence control flow at a later point in time by producing side effects.
- *Concurrent threads* — threads that alias a mutable object may concurrently modify its state and hence produce inconsistencies.

These abstractions provide common perspectives of studying the behavior and implementation of a program. For example, a software engineer frequently needs to understand which parts of a system implement a feature to carry out maintenance activities, as change requests and bug reports are usually expressed in terms of features [MEHT 02].

Compared to the two object dependencies caused by breaking encapsulation, the above listed kinds of dependencies are more complicated to analyze because they may not be detected in a single memory snapshot. For example, a feature may depend on another feature that was exercised at the very beginning of the program execution if object references are persisted between exercising the two features. Or object references may be transferred between threads if one thread assigns the object to a field and another thread later reads this field. Although object graph analyses can capture the class, feature, or thread in which a reference is created, they cannot relate this information back to arbitrary previous object graphs because they lack the notion of time.

Problem statement. *Because of the lack of analyses to detect hidden aliasing dependencies, object aliasing remains a major source of software defects and it is a hurdle for program comprehension.*

To analyze aliasing dependencies, we identify the following two dimensions in dynamic object-oriented program behavior.

- *Object reference structure* reveals how objects refer to each other at a given point in time (also referred to as object graph).
- *Object reference transfer* reveals where object references originate (also referred to as object flow).

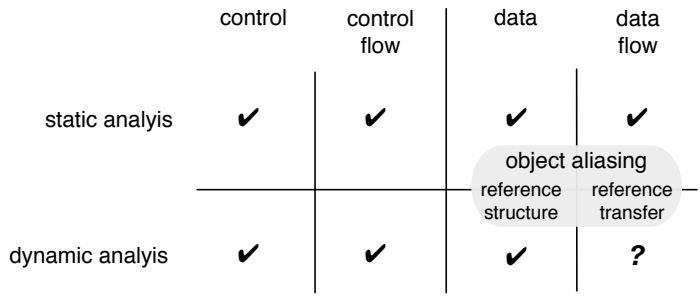

Figure 1.2: Reference structure and reference transfer dimensions of object aliasing mapped to data analysis and data flow analysis.

While the object reference structure is viewed as a snapshot, the object reference transfer dimension captures object flow during the execution of the program. An object reference transfer analysis is required, for example, to track how an object is passed from one field (instance variable) to another field via assignment — possibly being passed through methods and even arrays in between. This information can reveal dependencies between program elements that are located far in time and space, for example between the implementations of seemingly unrelated features or classes.

Figure 1.2 relates the two object aliasing dimensions to the program behavior categories of the Price *et al.* taxonomy [PRIC 93]. The reference structure dimension is captured by data analysis, and the reference transfer dimension is captured by data flow analysis.

Existing approaches capture data flow only to some extent. Dynamic data analysis cannot express reference transfer as it is based on models of object graph snapshots [DE P 00, DE P 02, HILL 02, MITC 06, FLAN 06, RAYS 06, PHEN 06]. In dynamic control flow analysis, method execution traces have been extended to carry information about objects involved in message sends or about the creation of objects [DE P 94, WALK 98, DEMS 02, GSCH 03, GOLD 05]. Also these extended models do not track the transfer of all references without gap.

A likely reason for the missing object-oriented dynamic data flow analyses is that execution tracing techniques originate in pre-object-oriented programming paradigms, like procedural programming and unstructured programming [RITC 93, CHEV 78]. The analysis of control flow was most important for such programs since they typically exhibited complex and tangled control structures (spaghetti code). In the object-oriented paradigm, however, complex object interrelationships are the new spaghetti code. With the shift to object-oriented programming, the analysis of object reference

transfer was forgotten by researchers who just adopted existing dynamic analysis techniques.

Research question. *How can we model object reference transfer?*

To answer this question, we need to develop a conceptual model to represent and reason about object reference transfer in a running system. A key criterion of our model is that it should also integrate existing dynamic analyses, such as control flow, since these dimensions of object-oriented program behavior are inherently intertwined.

1.2 Our Proposal: Object Flow Analysis

Thesis

> *Tracking object flow by explicitly representing object references and reference transfer is an effective way of analyzing and runtime monitoring dependencies introduced by object aliasing.*

We propose to explicitly represent object references in the dynamic analysis metamodel by a first-class entity called Alias. The transfer of object references is modeled by an association that captures the origin of an alias.

Figure 1.3 illustrates the core of the Object Flow Analysis metamodel. The entity Object represents any value in the system — both of reference type (objects and arrays) and primitive type (null, booleans, integers, etc.). All object references created at runtime, for instance when passing an object as parameter or when reading a value from a field, are represented by an Alias instance. The field of an object does not directly point to a value but to one of the aliases of this value. In other words, aliases introduce a level of indirection between objects, which allows one to exactly determine how objects are referenced at runtime.

The key strength of this model resides in the following relationships between aliases. The **origin** relationship models the transfer of object references. In Figure 1.3, the origin association of an alias is the alias that was used to create the alias in question. For instance, the origin of a field read alias always is a field write alias because going back one step in the flow of an object from the event of reading a field always leads to the event of writing the field. With this relationship we can precisely track how objects are passed through a system at runtime.

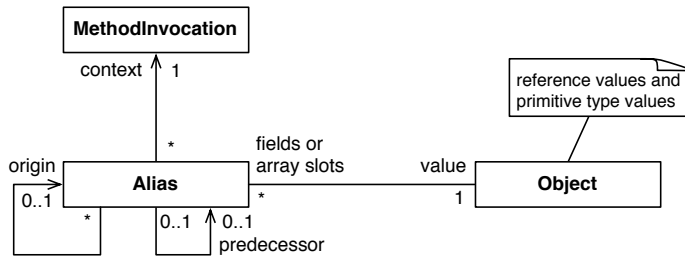

Figure 1.3: Core of the Object Flow Analysis metamodel.

The **predecessor** relationship is defined for field and array write aliases. It captures side effects to model the state history of objects and arrays. The predecessor of a field write alias is the field write alias of the value previously stored in the field. This relationship represents the reference structure dimension and it makes it possible to reconstitute the object graph as it existed at an arbitrary point of the program execution.

Another important aspect of our approach is the integration of the model of object references with the conventional method execution trace model (dynamic control flow). Each alias stores its creation **context** (the method invocation in which it is created), a timestamp, and the program counter to locate the position of its creation in the source code. Like this, it is possible to get detailed information about where and when an object reference is created, including the current call stack.

The goal of Object Flow Analysis is to provide a solid foundation for different kinds of dynamic object aliasing analyses. We claim that with our approach we can express software reverse engineering analyses that detect implicit dependencies introduced by object aliasing.

To validate our approach, we provide anecdotal evidence by proposing three analyses that detect different kinds of dependencies and make them explicit to support developers maintaining legacy systems. These three approaches analyze dependencies in classes, features, and in control flow. We do not propose analyses for dependencies in other program abstractions discussed above, such as object encapsulation, concurrent threads, or other static program entities than classes. These remain for future work.

Finally, to provide evidence that the concept of Object Flow Analysis extends beyond traditional applications in reverse engineering, we present an object-flow-aware virtual machine for back in time debugging. This virtual machine demonstrates that an implementation of the tracking and representation of object flow as proposed by our approach is an effective way of monitoring object aliasing at runtime.

1.3 Contributions

The main contributions of this dissertation are:

1. The concept and formal specification of Object Flow Analysis, a dynamic analysis model of data flow in object-oriented programs, which provides the foundation for a new category of analyses concerned with object aliasing.
2. A conceptual framework to reason about dependencies introduced by object aliasing and three reverse engineering approaches based on it that validate the usefulness of Object Flow Analysis.
3. The design and implementation of an object-flow-aware virtual machine for back-in-time debugging, which demonstrates that Object Flow Analysis provides an effective way of monitoring aliasing at runtime.

The following list details on the contributions of (2) and (3), which serve as a validation of our approach.

Visualizing Object Flow Between Structural Software Entities. Most existing dynamic analysis approaches focus on the execution of a program from the perspective of message passing [HAMO 04]. The contribution of our work on visualizing object flow is a novel, complementary perspective that reveals how classes and packages depend on each other by exchanging objects at runtime [LIEN 09].

Tracking Objects to Detect Feature Dependencies. There is a growing awareness amongst researchers of the potential of features in the context of reverse engineering. Much of the research in this area focuses on feature identification [ANTO 05], while only few researchers have investigated the dependencies between features [SALA 04]. Our work contributes to the state of the art by proposing a more accurate feature runtime dependency definition that takes aliasing into account. The additional dependencies that we uncover are precisely the indirect feature dependencies that can be problematic during maintenance [LIEN 07].

Exposing Side Effects to Support Writing Unit Tests. Writing unit tests for legacy systems is a key maintenance task — but if developers lack internal knowledge of the system, this task is non-trivial [DEME 02]. To implement a fixture and assertions, the developer has to understand what objects the unit under test depends on and what the expected side effects of the unit's execution are. To address this problem, we propose an object reference analysis that exposes side effects in control

flow, and that uses these side effects to guide the developer when writing tests [LIEN 08a].

Practical Back-in-time Debugging. Two major downsides of back-in-time debugging have been its high memory consumption and significant performance impact [POTH 07]. We propose an approach that features significant improvements on both accounts. Its underlying idea is to capture execution history not as a trace of events but to leverage object references to first-class objects in the virtual machine. This work makes an important contribution to the state of the art in back-in-time debugging [LIEN 08b].

1.4 Structure of the Dissertation

Chapter 2 discusses the state of the art in dynamic data analysis to identify the extent to which existing approaches capture relevant data for analyzing object aliasing. This survey shows that there is a gap in tracking object reference transfer.

Chapter 3 introduces Object Flow Analysis, our approach to a dynamic analysis of object reference transfer. We show how the proposed metamodel also captures object reference structure and how it relates object references to dynamic control flow. We provide a formal specification of how objects are tracked at runtime and we introduce a conceptual framework based on our metamodel to reason about dependencies introduced by object aliasing.

Chapter 4 presents a visualization of object flow that supports the reverse engineering process to discover dependencies between structural software. The interactive views allow the developer to iteratively discover the flow of objects between classes and packages.

Chapter 5 proposes an accurate feature dependency definition and a detection strategy based on the Object Flow Analysis metamodel. We also propose a visual approach to support developers interpret feature dependencies.

Chapter 6 presents Test Blueprints, an analysis of object transfer in method invocations that helps the developer to write unit tests for an unknown system. The Test Blueprint of a selected part of the program execution serves as a plan to write a new unit test. It displays detailed information about the fixture and the expected side effect of running the test method.

Chapter 7 discusses a back-in-time debugging approach that tracks execution history by capturing object references in the memory model of the virtual machine. We demonstrate that the Object Flow Analysis metamodel captures key relationships between object references — and as a consequence, irrelevant history is automatically garbage collected.

Chapter 8 concludes the dissertation and ends with an outlook on the future work opened by our approach.

Chapter 2

Approaches to Dynamic Data Analysis

In recent years, there has been an increasing research effort dedicated to the dynamic analysis of object-oriented software. A coarse classification of approaches can be made based on whether an approach analyzes dynamic control flow (also referred to as method execution tracing) or dynamic data (object graph structure analysis). Hybrid approaches also exist, for instance for the implementation of back-in-time debuggers where both the control flow and data history are required to move a debugger backwards to any previous point of the program execution.

The two perspectives taken by control flow and data analysis seem like a natural choice for analyzing the behavior of an object-oriented system. The reason is that message passing and reference semantics are two cornerstones of object-orientation. Objects collaborate to accomplish a complex task and this collaboration is expressed through the exchange of messages that are sent along the path of object references. Dynamic control flow and data analyses capture this runtime behavior.

However, this view is not complete because it fails to show how object references are established in the first place. Indeed, computation in an object-oriented program essentially is the process of *transforming* the graph of objects, and this transformation is only possible by passing around object references. This means, to make one object point to another object in the object graph, an existing reference has to be transferred to it (or a new one has to be created by instantiating a class).

Object aliasing introduces complex dependencies that determine the behavior of the program. To study these dependencies, we need a solid

foundation to express object flow — for instance to provide more powerful debugging tools or to detect runtime dependencies between the features of a software system.

In the introduction chapter we argued that to capture object aliasing we need to analyze both the reference transfer and the reference structure of objects. To make this statement more concrete, let us consider a debugging session in which we need to figure out the following information about a field of some object:

- *How was the current value passed into the field?* This question can be answered by tracking the **origin** of the value stored in the field.
- *Which values were previously stored in the field?* This question can be answered by tracking the state **history** of the field.
- *Where in the control flow was the current value read?* This question can be answered by tracking each **context** in which the field was accessed.

The first aspect, **origin**, captures the reference transfer (object flow), whereas **history** captures the reference structure of objects. The last aspect, **context**, captures in which method invocation the references are created; it links the previous two aspects to the runtime control flow.

In this chapter we review the state of the art in dynamic data analysis. Approaches exist that, to some extent, capture dynamic information about origin, history, and context. We divide the approaches into the following three major categories.

- Dynamic Data Structure Analysis, discussed in Section 2.1, is concerned with the shape of object graph snapshots. While these approaches show the aliasing relationships between objects, they do not provide information about where and in which context a reference in this graph originates. To reconstruct arbitrary intermediate object graphs, some approaches track the history of fields.
- Dynamic Data Flow Analysis, discussed in Section 2.2, captures runtime data flow in relation to static program elements. These approaches provide partial origin information by keeping track of the connection between writing and reading a variable. Context information is limited to the location in the static control flow graph.
- Extended Execution Trace Analysis, discussed in Section 2.3, tracks the origin of objects passed as parameters and return values, and the allocation of objects, and keeps context information, such as the target and caller of a method invocation.

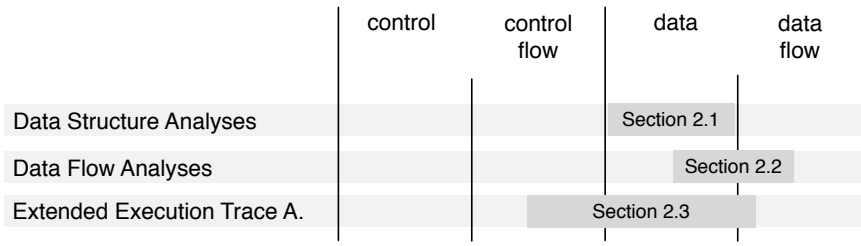

Figure 2.1: Categories related to Dynamic Data Analyses.

Figure 2.1 puts these three categories into a broader context. Focusing on these three categories we limit the scope of our literature review to the approaches related to dynamic data flow analysis.

Structure of the chapter. The first three sections discuss the state of the art in Dynamic Data Structure Analysis (Section 2.1), Dynamic Data Flow Analysis (Section 2.2), and Extended Execution Trace Analysis (Section 2.3). Section 2.4 concludes the chapter.

2.1 Dynamic Data Structure Analysis

Tool support to help understand object reference graphs is particularly important for programs that make extensive use of heap based data structures, such as in Java and in other object-oriented languages. Several tools exist for the manual inspection of heap memory snapshots (Section 2.1.1). More advanced approaches, which we refer to as *shape analyses*, are concerned with the connectedness and entry points of object graph structures (Section 2.1.2). Shape analysis techniques reason about object graph structures with the goal to identify ownership [HILL 02], to optimize garbage collection [SHAH 00], to support debugging [RAYS 07], or as part of a general understanding of program behavior [PHEN 06].

2.1.1 Heap Snapshot Browsers

The Heap Analysis Tool (HAT)[1] from Sun allows the programmer to create heap snapshots using the JVMPI (Java Virtual Machine Profiler Interface), which produces a complete dump of all the objects in the Java heap at a

[1] http://hat.dev.java.net/

given time. HAT supports queries such as "show all instances of class X", "show all referrers to a specific object", "show all objects reachable from a specific object". This tool can help to debug and analyze the objects in a running Java program.

Similar products are Quest Software's JProbe Memory Debugger[2] and Codework's Jprofiler[3], which provide browsable views of the web of objects in a heap snapshot. Other tools use visual representations to browse memory, for instance the GNU Data Display Debugger [ZELL 96].

Although these tools provide a means to directly browse the object graph, they do not provide higher-level views or an analysis of structural properties.

2.1.2 Shape Analysis

The following more sophisticated approaches help to investigate object graphs to analyze the shape of object structures in memory snapshots. Such fully or semi-automated analyses are used to identify patterns, such as object encapsulation and ownership, or to help finding the causes of memory leaks.

General reference pattern analysis. Jinsight, a freely available tool that is part of IBM's Rational Application Developer for WebSphere Software, provides various dynamic analyses [DE P 99, DE P 00, DE P 02]. The analysis of reference patterns supports developers to find the causes of a memory leak by comparing two snapshots of all objects and references, one taken before performing a critical action and one taken afterwards. The focus is on finding bad references from previously existing objects to new objects. The extraction of patterns reduces complexity by aggregating repetitions in object structures.

Pheng and Verbrugge identify trees, directed acyclic graphs, and cycles in dynamic data structures of Java programs [PHEN 06]. Rather than taking memory snapshots they use field write event traces gathered through the JVMPI. This trace of events is then fed into an analyzer that reconstructs heap snapshots for every n-th event. This approach provides more flexibility concerning the point of time and number of snapshots taken. However, the number of snapshots is limited because of the space and performance characteristics of the applied shape analysis.

A similar technique was adopted by Flanagan and Freund to extract object models by reconstructing each intermediate heap from a log of object allocations and field writes [FLAN 06]. The analysis applies a sequence of

[2] http://www.quest.com/jprobe/

[3] http://www.codework.com/jprofiler/product.htm

abstraction-based operations to each heap, and combines the results into a single object model that conservatively approximates all observed heaps from the program's execution.

Dynamic ownership inference. The notion of *object ownership* is most commonly understood as the graph-theoretic dominance tree of heap object reference relations. In the heap referencing relation there is an edge from object x to object y iff some field f of object x refers to object y at some time.

Visualizations of ownership trees proposed by Hill, Noble and Potter show the encapsulation structure of objects [HILL 00, HILL 02]. The goal of their research is to extract a program's implicit encapsulation structure from its object graph. With respect to object flow, this approach is interesting. It has to deal with the difficulty that the transfer of objects when they change ownership is hard to grasp by looking at two subsequent snapshot views. Noble *et al.* propose to use animations to help the user follow the transition changes between two ownership visualizations.

Mitchel implements a dynamic ownership inference for detecting inefficiencies in a program's memory footprint for very large heaps [MITC 06]. This analysis uses a single heap snapshot taken at a critical moment in time. The main contribution of this work is the identification of four common graph structure patterns to identify potential memory leaks.

Rayside *et al.* propose an analysis that reveals object ownership and sharing in a hierarchical matrix [RAYS 06]. In later work, Rayside *et al.* propose a technique for finding complex memory leaks through a dynamic ownership analysis [RAYS 07]. Their tool computes an abstracted ownership tree from an object reference graph. Allocation time and size data are aggregated up the ownership hierarchy and plots of this data are generated to identify potential memory leaks.

Dietl and Müller dynamically infer universe ownership type annotations from traces of Java programs [DIET 07]. They build a cumulative representation of the object graph by tracing all objects that ever existed in memory, all references between these objects, and which objects modified which other objects. In their model, they distinguish two types of references: (i) write references for field assignments and calls to methods producing side effects, and (2) so-called naming references for reading fields and calling side effect free methods.

2.1.3 Summary

The analyses discussed in this section focus on properties and the shape of object graph snapshots. Some approaches do not just analyze memory dumps but they trace each program state modification, and some approaches additionally record reference read access.

While these approaches reveal the aliasing relationships between objects and some of them capture the state history of the program, they do not provide information about where an object reference originates, nor do they provide context, like the current call stack, to map state changes to control flow.

2.2 Dynamic Data Flow Analysis

In contrast to the previously discussed analyses of object graph structures, this section discusses approaches that analyze the flow of values. This section is divided into three groups of approaches. The first group is concerned with analyzing access sequences on variables, and the second group is concerned with analyzing the control flow related to a selected object. The third group tracks data flow in and between methods to identify methods that are free of side effects (that is, the only visible effect of their invocation is to return a value).

2.2.1 Define-Use Analysis

Define-use Analysis is a method to analyze the sequence of actions on variables. An assigned value *flows* into the computation (definition site) and later it flows out when being used (usage site). A variable gets undefined either when assigning null or when it goes out of scope. The goal of this analysis is to detect improper sequences on data access for testing and debugging [CHEN 95, BOUJ 00].

The technique originates in compiler optimization and has primarily been applied to testing procedural programs. More recently, Andrew Cain proposed an extension of the technique tailored to object-oriented programming languages [CAIN 05].

These approaches are often also referred to as dynamic data flow analyses. Originating in static analysis, the term *data flow* in this context refers to actions on individual variables. Hence, the focus is on the flow of values into and out of variables rather than on the complete flow of values through a system.

2.2.2 Dynamic Program Slicing

The define-use analysis approaches give only limited information on how the recorded actions relate to control flow. While the define and use sites are known, no information is gathered about how control flow progressed from one site to another or how a variable influenced the flow of control. In contrast, program slicing [WEIS 81] uses both static control and data dependencies to identify all expressions that influence a given variable at a certain source location or are influenced by this variable. Dynamic slicing further reduces the graph by only taking statements into account that have been executed in a concrete execution scenario [KORE 98].

Recently, Object Process Graphs (OPG) have been proposed that provide even smaller slices. OPGs are sparse control flow graphs that contain only nodes relevant to a selected value [EISE 05a]. This means, an OPG shows control dependencies, loops, procedure calls, and read/write operations in which the value is used but it excludes statements that have dependencies to these nodes. This analysis was first carried out statically [EISE 05a], and later it has been implemented as a dynamic analysis [QUAN 08], which for example was used for protocol recovery [QUAN 07]. Dynamic OPGs do not differentiate object references since all operations are mapped to the static control flow graph. Therefore, dynamic OPGs do not reveal the flow of objects but rather the flow of control between points in the execution at which the object was used.

2.2.3 Side Effect Analysis

A method is considered to be free of side effects if the invocation of the method and all indirectly invoked methods do not modify any externally visible object. This means, even if objects are modified in a method but this modified state is not accessible anymore after the method has returned, the method is not considered to produce side effects. This analysis has mainly been performed statically; it originated in the area of compiler construction 30 years ago [BANN 79]. Recently, a few dynamic side effect analyses for object-oriented languages have been proposed, which complement static analyses. They take advantage of dynamic data for better scalability and to alleviate the very conservative approach typically taken by static analyses.

JDynpur[4] is a dynamic side effect analysis tool for Java developed by Dallmeier. The tool records method start and end timestamps, instantiations and field writes through a bytecode instrumentation. JDynpur also dynamically analyzes parameter reference mutability, which classifies parameters of a method as either immutable or mutable.

[4] http://www.st.cs.uni-saarland.de/models/jpure/

A scalable and accurate parameter reference mutability analysis has been proposed by Artzi *et al.* [ARTZ 07]. Their approach composes static and dynamic analyses to stepwise refine the results. A similar work, which also combines static and dynamic analysis, was proposed by Xu *et al.* [XU 07a]. As an application, they use their approach to automatically cache method invocation results of pure methods.

2.2.4 Summary

The approaches discussed in this section capture the flow of objects and values in relation to the static program elements. For example, by keeping track of where in the static control flow graph a variable was written and where it was read, these approaches partly track an object's flow; detailed information about object references, however, is missing. This means for example, that the program state is not modelled and hence no links between the available origin information and object states can be drawn. The strong relationship to the static program analysis shows the historical origin of those approaches. All of them have first been proposed in static analysis and have now been implemented as dynamic analyses to complement their static counterparts.

2.3 Extended Execution Trace Analysis

Apart from the dynamic data analyses discussed above, a major group of dynamic analyses investigates the runtime control flow of a program based on method execution traces [HAMO 06, KLEY 88, DE P 98, RICH 02, GREE 05, ZAID 05]. This group of approaches is based on the assumption that a program's execution can be characterized as a succession of interesting events [DE P 94]. Most tracing techniques introduce sensors into methods, which generate an event when being encountered. Typically, the call trees are later reconstructed using the start and end timestamps or the stack depth recorded with each method invocation event. Instrumentation is usually achieved by manipulating the bytecode of the target program [DAHM 99, CHIB 03, BRUN 02, DENK 07]. Recent approaches have also made use of the JVMTI (Java Virtual Machine Tool Interface, part of the Java 5 platform) and of aspect weaving [GSCH 03, ZAID 06]. In Smalltalk, wrapping method objects to intercept method execution is a widely used technique [BRAN 98].

2.3.1 Trace-based Reverse Engineering Approaches

While this group of approaches concentrates on the dynamic control flow by reconstructing the sequence and nesting of method executions, some approaches additionally incorporate information about the objects involved.

Gschwind and Oberleitner propose to trace also the parameters for each method invocation event to increase the detail in UML sequence diagrams [GSCH 03]. They note that "although we initially thought that accessing the parameters [...] is just a nice little gimmick, we were surprised that this functionality was of crucial importance for reverse-engineering". The reason is that concepts in the software are not only encoded in the target object and in the signature of methods, but also in the objects passed between methods. In the case study presented by Gschwind *et al.*, they investigated a particular feature of a program but in the first version of their approach the execution trace did not contain the class in question because the objects passed as parameters were not identified.

Another datapoint captured by extended execution traces is the creation (and destruction) of objects [DE P 94, WALK 98, GSCH 03, GOLD 05]. For example, De Pauw *et al.* developed a number of visualizations, such as instance histograms and allocation matrices, to show fine-grained information about objects. For example, visualizations show the number of instances grouped by class at a given time. Sub-views reveal detailed information like the number of instances of a class created by methods of another class.

Walker *et al.* visualize the operation of a system at the architectural level [WALK 98]. By means of a declarative mapping language the user of their tool can create so-called cells that are represented as boxes to visualize the interaction of messages between different structural parts of the application. A cell also presents statistics about creation and destruction of objects, however, the propagation of objects *between* cells is missing. Indeed, they note that "it may be useful to capture the migration of objects if an object is created in one subsystem, but is then immediately passed as an argument to another subsystem".

Demsky and Rinard analyze execution traces that include object aliasing events to synthesize a set of conceptual object states (which they call "roles") [DEMS 02]. The goal of their analysis is to help developers understand heap properties of object-oriented programs and how the actions of the program affect these properties. An interesting aspect of their approach is that the analysis also maintains a set of inverse references in addition to reconstructing the heap. For each reference in the original heap, in their model there exists one inverse reference. This model allows their analysis to quickly find the source of a reference and the field containing the reference.

2.3.2 Complete Execution History Recording

Back-in-time debugging. The most common approach to implementing back-in-time debuggers has been to create a trace log of the program execution. In this execution trace, also data about side effects has to be logged so that any past object state can be reconstructed. Moreover, parameters and return values of method invocations are recorded. A representative set of event types are: field write, local variable write, array write, method call, method enter, and method exit. In this model — taken from TOD, a back-in-time debugger recently proposed by Pothier *et al.* [POTH 07] — each event has a set of attributes. For example a field write event has a timestamp, thread id, source code location, field id, value.

Other back-in-time debuggers use very similar tracing techniques and event models. A popular tool is ODB, a back-in-time debugger for Java proposed by Lewis *et al.* [LEWI 03]. Unstuck is a similar proof of concept implementation for Squeak Smalltalk [HOFE 06], and Omnicore's CodeGuide[5] is a commercial back-in-time debugger. While TOD stores the events in a distributed database for improved scalability, the other three approaches keep the complete event history in memory.

Query-based debugging. Related to back-in-time debugging is query-based debugging. In these tools the user formulates a query in a higher-level language, which is then applied to the execution history [MART 05, LENC 97, POTA 04, DUCA 06]. Queries can test complex object interrelationships and sequences of related events. Approaches exist that execute the query at runtime, which can improve performance because no history has to be stored [LENC 99, GOLD 05].

2.3.3 Summary

The execution tracing approaches focus on the analysis of the runtime control flow. We have selected the approaches that also keep track of the objects involved in method invocations by recording the objects used as targets, parameters, and return values. Like this, approaches capture origin information related to the transfer of object references between methods. Hence also the context in which these w take place is known. However, the flow of objects below the granularity of methods is not captured. For example, the links between the events of passing an object as parameter, storing it in a field, and later accessing it from within a different method are not drawn. Hence, the models proposed by these approaches allow only for limited origin tracking because they leave gaps in the flow of an object.

[5] http://www.omnicore.com/

2.4 Conclusion

In the remainder of this chapter we illustrate the missing origin aspect in the discussed approaches on an example taken from a Smalltalk bytecode compiler. Tracking the origin of an object reference is motivated by the question: "why is the field pointing to value x?". In other words, we want to be able to precisely know how an object reference is passed from one point in the program execution to another one.

Control flow view. Method execution traces, which are used to analyze the runtime control flow, are usually represented as a method call tree. Figure 2.2 illustrates a small excerpt of a trace of the compiler, displaying the method executions as a tree (the notation follows the pattern TargetClass »methodsignature). By reading the trace in Figure 2.2 we see that an IRMethod instance is created in IRBuilder by a call of the new primitive. Further down in the trace, an IRMethod instance is sent the message compiledMethod in the method RBMethodNode»generate.

With the data gathered by extended execution tracing approaches we can reveal that the IRMethod object is the identical instance in both places of the trace. However, we cannot reliably answer how this instance was passed from where it is instantiated to where it is used later on. The instance could be passed from IRBuilder to RBMethodNode via a sequence of method return values through other classes, but it could just as well be stored in a field and then be accessed later on (or a combination of both scenarios).

Speculating about the answer is further hampered by the sheer size of execution traces. Figure 2.2 only shows the 6 levels and 8 method executions. In our case, though, the area of the tree hidden by the dots is 46 levels deep and comprises 4793 method executions.

Data structure view. With the approaches categorized as dynamic data structure and data flow analyses we can precisely reconstruct any intermediate object graph snapshot to investigate how objects are aliased.

Object graphs can be visualized as UML object diagrams [FOWL 03] as illustrated in Figure 2.3. This figure highlights a small part of the object reference graph taken from the same execution as the one illustrated in Figure 2.2. The snapshot is taken just before IRMethod»compiledMethod is executed. In Figure 2.2 we can see that IRBuilder references the IRMethod instance (from the trace we know that the IRMethod instance is created in IRBuilder). Yet, like with the control flow perspective, the data centric perspective does not reveal how the IRMethod instance is passed from the IRBuilder to the RBMethodNode. The missing link is how the object references are transferred.

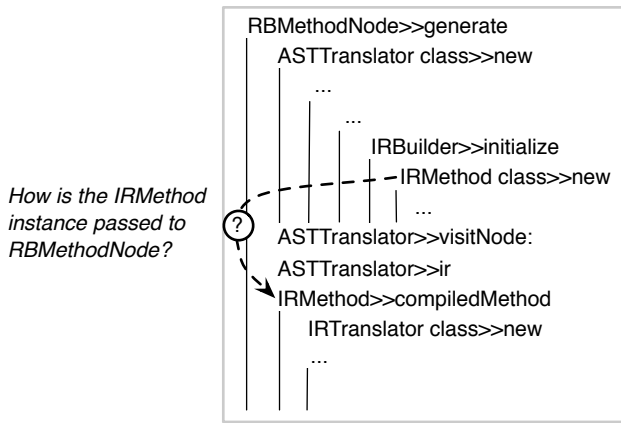

Figure 2.2: Excerpt of an execution trace represented as a call tree.

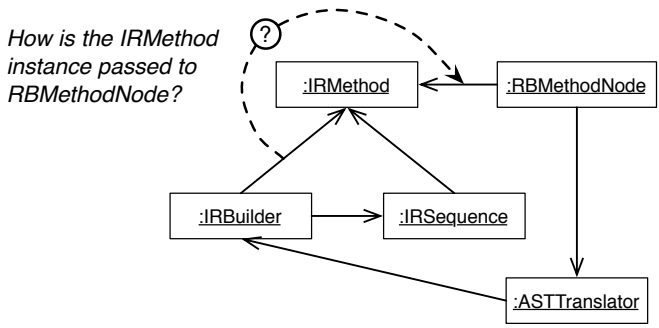

Figure 2.3: UML object diagram representing an excerpt of an object graph.

From our literature review we can conclude that the different bits and pieces required to track the origin of objects, the history of program state, and their context in the program execution are actually all captured in one way or another. But no solution has been proposed so far that combines those datapoints and draws the missing links to get a model that can precisely and completely express dynamic data flow in object-oriented systems.

Chapter 3

Object Flow Analysis

Pervasive aliasing in object-oriented systems complicates program comprehension and remains a major source of software defects. Yet, as our literature survey has shown, the state of the art in dynamic analysis lacks the concept of object reference transfer. In this chapter we present our approach, which we refer to as Object Flow Analysis. The goal of our approach is to provide a coherent model of object reference transfer that provides a basis on which different kinds of object aliasing analyses can be defined.

Structure of the chapter. In this chapter we introduce Object Flow Analysis, our approach to dynamic data flow analysis, which is the foundation on which the work presented in the following chapters build. In Section 3.1 we introduce the Object Flow Analysis metamodel, which defines how object flow is *represented*. In Section 3.2 we provide a formal specification of Object Flow Analysis, which defines how to *observe* the flow of objects at runtime to generate a model that conforms to the defined metamodel. Having defined how object flow is observed and represented, in Section 3.3 we introduce a small framework that provides a basic conceptual structure to *reason* about dependencies introduced by object aliasing. Section 3.4 concludes this chapter and provides an outlook to the subsequent chapters of this dissertation.

3.1 The Object Flow Analysis Metamodel

The underlying principle of Object Flow Analysis is to explicitly represent object references and to capture the relationships between references.

Figure 3.1 illustrates the core of the Object Flow Analysis metamodel. It integrates entities typically used in dynamic analysis metamodels, such as Object, MethodInvocation, Method, and Class. These existing entities are shown in gray, whereas the new entities and associations introduced by the Object Flow Analysis metamodel are shown in black.

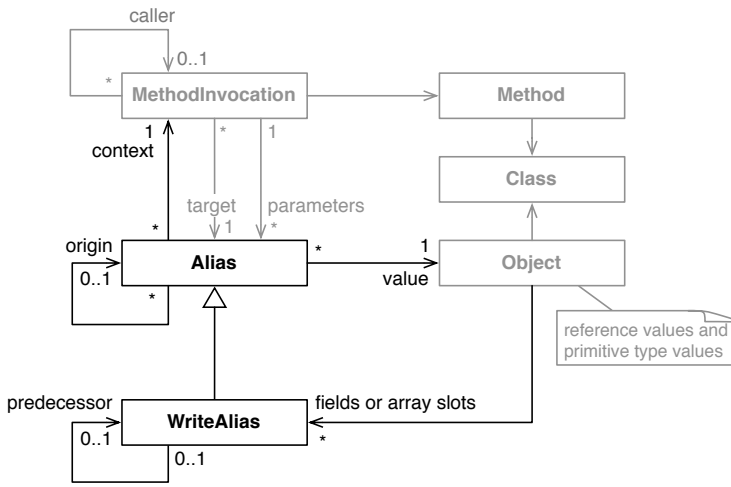

Figure 3.1: Object Flow metamodel (gray entities and associations indicate objects present in typical dynamic analysis metamodels).

Object references are represented by the entity Alias. (We chose the term *alias* to distinguish it from the term *object reference*, which is the concept it represents.) Each situation in which an object reference is transferred is captured by an alias. Each alias points to exactly one object, the object to which the reference points. The entity Object represents any value in the system, both of reference type (objects and arrays) and of primitive type (depending on the execution environment the undefined value, booleans, integers, etc.).

Aliases introduce a level of indirection between objects referencing each other. In our model, the field of an object does not directly point to another object, but rather to a write alias of that object. Analogous, each array slot points to a write alias. The same holds for objects passed as parameters, objects being returned from method invocations, and objects being read from fields and arrays. The target of a method invocation also points to an

alias, the alias through which the message is sent to the object. By recording all aliases of an object, we can exactly determine how objects are referenced at any point in the program execution.

In Figure 3.2 we zoom in on the entity Alias and illustrate the class hierarchy of the eight kinds of aliases in our metamodel. We classify these aliases in groups of two as follows.

CreationAlias represents an object reference created when:

1. an object is instantiated or cloned (referred to as *allocation alias*)
2. a literal object is referenced (*literal alias*)

MethodAlias represents an object reference created when:

3. an object is passed as a method parameter (*parameter alias*)
4. an object is returned from a method invocation (*return alias*)

ReadAlias represents an object reference created when:

5. an object is read from a field (*field read alias*)
6. an object is read from an array (*array read alias*)

WriteAlias represents an object reference created when:

7. an object is written into a field (*field write alias*)
8. an object is written into an array (*array write alias*)

The rationale is to capture all situations in which an object is made visible in a method invocation (1–6) and in which a side effect is produced (7, 8).

AllocationAlias represents references of newly created objects when they are passed to the application from the new primitive. LiteralAlias instances are created in our model when a literal object is referenced at runtime (*e.g.*, when referring to the literal true in the source code).

Whenever an object is assigned to a field or to a slot of an array, a new write alias is created. A special case of field and array assignment is the initialization of the fields and of the array slots with null when an object is instantiated or an array is allocated. Although these assignments are not directly visible in the source code, instances of FieldWriteAlias and ArrayWriteAlias are created for it. This is important for a complete tracking of the flow of null.

We do not capture writing into and reading from local variables and make the flows through them transparent. From our experience with applications built on top of Object Flow Analysis, local aliases have not added value as they denote only reference transfers within a method (extending the model with local read and write aliases, however, would be straightforward).

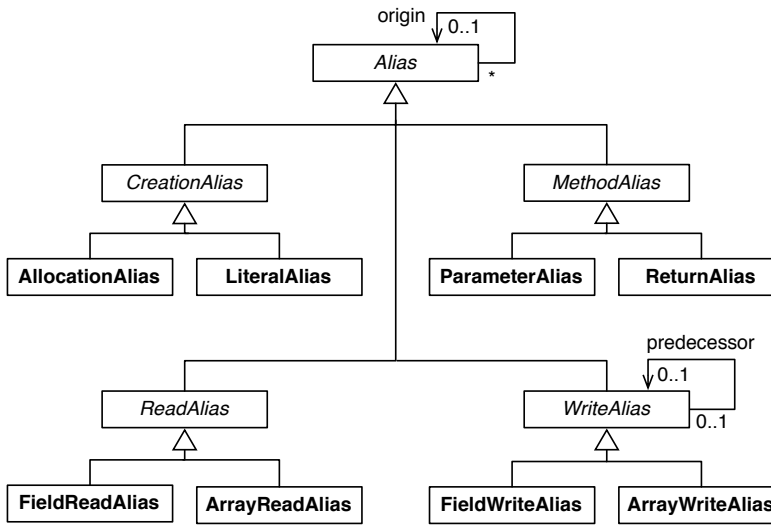

Figure 3.2: Alias class hierarchy.

The key strength of the Object Flow Analysis metamodel resides in the following three associations between aliases and method invocations (see Figure 3.1):

- The origin relationship between aliases models the object reference transfer. The origin of an alias is the alias that was used to create the alias. For instance, the origin of a field read alias always is a field write alias because going back one step in the flow of an object from a field read alias always leads to the field write alias. Except for the creation aliases, each alias in the model has exactly one origin alias, and any alias can be the origin of potentially many other aliases. Creation aliases mark the beginning of an object flow.
- The predecessor relationship, which is defined for WriteAlias, models the history of program state. The predecessor of a field write alias is the field write alias of the object previously stored in the field (analogous for array slots).
- The context relationship between aliases and method invocations captures where in the control flow an alias is created. It relates object flow to control flow.

In the following three sections we detail on the origin, predecessor, and context relationships.

3.1.1 Origin Relationship

Apart from the creation aliases, all aliases originate from a previously existing alias. This gives rise to the *origin* relationship between the aliases of an object. The origin of an alias is the alias that was used to create the alias in question. Organizing the aliases of an object by their origin relationship forms a tree. Such a tree represents the flow of an object, and we therefore refer to it as *object flow tree*. With the explicit representation of references and their origins, the Object Flow Analysis metamodel completely captures the flow of objects through a system.

Except for literal objects, each object has exactly one object flow tree, with the root alias being the allocation alias created by the instantiation or clone primitive. Object flow trees of null, true, false, and other literal objects have a write alias or literal alias as their root node. These objects usually have multiple object flow trees since they can occur independently in different places of the program execution.

Example. To illustrate details of the object flow construction, we use a Smalltalk bytecode compiler as a concrete example. The compiler has four phases: (1) scanning and parsing of source code, (2) verifying and annotating the abstract syntax tree (AST), (3) translating AST to the intermediate representation (IR), and (4) translating IR to bytecode.

The example used in this section shows the interplay of important classes of the last two compiling phases (translating the AST to the IR, and translating the IR to bytecode). We focus on an instance of the class IRMethod which represents a method in the IR. It acts as a container of IRSequence instances, which group instructions and form a control graph.

Figure 3.3 lists the relevant Smalltalk source code and below illustrates the object flow tree, which represents the object flow of the IRMethod instance. The dashed boxes represent instances of the entity Alias. An arrow indicates the origin relationship between two aliases. Note, since the arrows point from an alias to its origin, they point in the opposite direction of the actual flow of the object.

The flow of the IRMethod instance starts with the root alias *allocation* (1) in the method IRBuilder>>initialize where the instance is created. The object is then directly assigned to a field named ir, which is represented as a field write alias (2).

During the life cycle of IRBuilder the object is read from the field in (3) and then passed as parameter of method: (4) to IRSequence objects where it is stored in a field called method. In the actual execution we analyzed, the branch (3-5) was created multiple times because for each new sequence

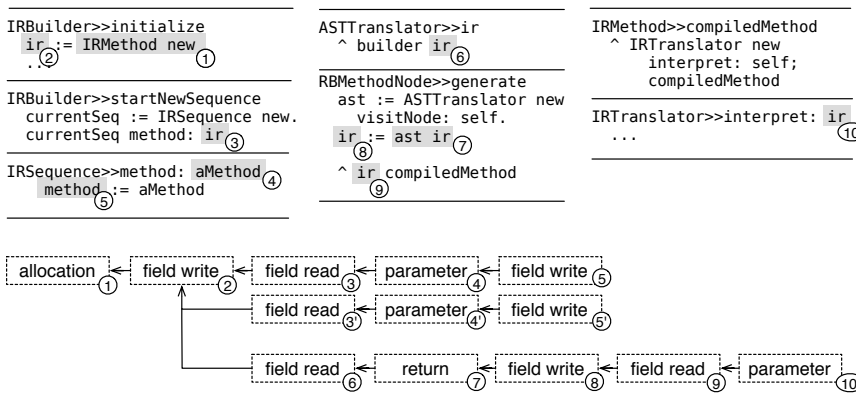

Figure 3.3: Object flow of an IRMethod instance in a bytecode compiler.

instantiated by the method startNewSequence, the IRMethod object is passed to it.

When RBMethodNode>>generate requests the IRMethod instance, the object is first read from the field in IRBuilder (6) (this happens through a getter, which is omitted for conciseness). And only afterwards it is returned from ASTTranslator to RBMethodNode (7). Being returned, the object is directly stored in the field ir of RBMethodNode (8). In the same method the field is read to send the message compiledMethod (9).

An interesting aspect of our model can be observed in the last step of the object flow tree illustrated in Figure 3.3. In compiledMethod an IRTranslator is instantiated and then the target object passes itself as parameter of the message interpret: to this new object. The origin of the parameter alias (10) is the field read alias that was used to send the original message compiledMethod.

There is no alias created for self, the target of the method invocation. Rather, when passing self, the origin of the new alias is the alias that was used to refer to the target object when invoking the current method. The rationale is that there is no reference transferred when sending a message to an object. Only when passing the current target of a method execution, is a reference transferred. Consequently, the origin alias of the newly created alias is the reference through which the message is sent to the target object. Therefore, all aliases — except for the creation aliases — have an origin. This property assures that the object flows do not have gaps.

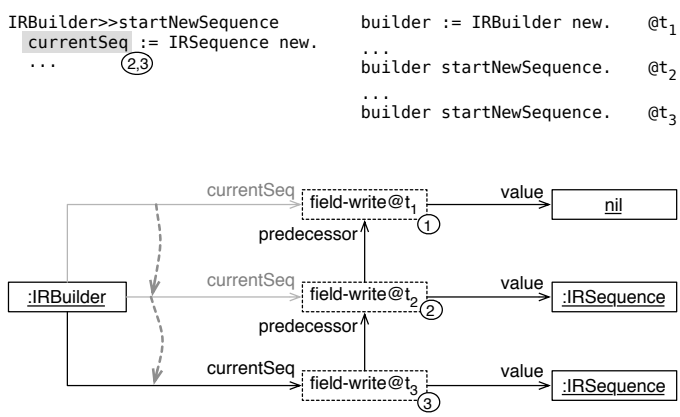

Figure 3.4: Capturing historical state through the predecessor relationship.

3.1.2 Predecessor Relationship

The predecessor relationship is only defined for write aliases, which capture side effects. While the origin relationship models the dimension of the flow of objects, the predecessor dimension models the history of the program state. The purpose is to capture the modifications of fields to be able to determine the value of an object's field (or the value at a specific index of an array) at an arbitrary point of the program execution.

The predecessor association of a field write alias is the field write alias of the value previously stored in the field (respectively the array write alias previously stored in the slot of the array). Figure 3.4 shows an example of an IRBuilder instance with attribute currentSeq that undergoes two state changes. In this instance diagram, solid boxes represent application objects and dashed boxed indicate write alias instances. When the object is instantiated at the point in time t_1, the field is initially undefined (it points to the undefined object nil). This situation is represented by a field write alias (1), which is created when the object containing the field is instantiated. Later at t_2, a first IRSequence instance is assigned to the field (2), and at t_3 a second one is assigned to the field (3) in the method startNewSequence.

In the example, the field first points to the alias of null, then to the alias of the first sequence object, and last to the alias of the second sequence object. Except for the first alias, each subsequent write alias of the field keeps a reference to its predecessor, the alias that was stored in the field beforehand. In this way, the alias pointed to from a field is the head of a linked list of aliases that constitute the history of that field.

With this information we can answer the question "When did the field point to another object?". Reconstructing the object graph and all object states at a selected point in time is straightforward with this model. For each field the predecessor list is traversed backwards starting at the last alias to find the most recent write alias before the selected timestamp. In the example in Figure 3.4, accessing builder.currentSeq at timestamp t_3 directly returns the alias of the last sequence, whereas at t_1 the alias instance pointing to nil is returned.

3.1.3 Context Relationship

The context association of an alias is used to navigate to the method invocation in which the alias was created, or vice versa, to find all aliases created in a method invocation. Each alias is created in exactly one method invocation, the invocation in which the object is made visible or in which a side effect is produced. Consequently, the context of a parameter alias is the method invocation to which the parameter is passed. Analogously, the context of the return alias is the method invocation to which the object is returned.

The entity MethodInvocation represents a frame on the call stack. To model the call stack, each method invocation holds onto its caller. Moreover, each alias instance stores a creation timestamp and the source code location (program counter) to keep track of where exactly it was created inside of a method. The context relationship links the object flow with the control flow. This provides precise information about where and when an object reference was transferred. *Where* means that we can retrieve the complete call stack at this point in time. *When* means that we can reconstruct the object graph at that point in time. On the other hand, as the target of a method invocation also points to an alias, it is possible to determine which messages are sent to an object through a specific object reference.

3.2 Specification of Object Flow Tracking

While the first part of this chapter informally presented the conceptual model of Object Flow Analysis, in this second part we provide a formal specification. The motivation for this specification is that the informal description alone makes it hard to accurately understand our approach and to reproduce its results by an independent researcher.

Our informal description lacks precise information about how the dynamic data is captured at runtime and how the model is built in the first place. As an example, consider the field write alias that is created when initializing a field with null. Which method invocation is the context of this alias?

We can find similar limitations in other publications. For instance, Quante *et al.* describe only superficially how in their Object Process Graph approach data are gathered from C programs, leaving room for different interpretations [QUAN 08]. The following two sentences offer an example for such possible interpretations. "By object, we mean a local or global variable or a variable allocated on the heap at runtime. Hence, we consider a trace a sequence of operations applied to an object." [QUAN 06]. On the one hand talking about variables allocated on the heap is rather ambiguous (*e.g.*, does this include primitive type values?). On the other hand it is not clear which operations on those objects are meant, as operations could mean to change the state of an array or record, passing pointers of a value, or changing the value of a variable.

To concisely specify how aliases are recorded from analyzing the execution of an object-oriented program, we define a minimal object-oriented imperative language with standard semantics. By extending the reduction rules, we then define the creation of aliases that constitute a model conforming to the Object Flow Analysis metamodel.

This language is influenced by established approaches [CAME 07, CLAR 02, DROS 08], and is similar to recent work of Drossopoulou *et al.*, who propose a formal framework to specify object invariants [DROS 08]. The language supports imperative features, including a heap, field assignement, and it explicitly represents call stack frames.

The main difference in the syntax and reduction rules compared to the approach by Drossopoulou *et al.* [DROS 08] is that we unify primitive type values and reference values. In our language, we refer to primitive values, such as null, in runtime expressions through addresses. That is, the value null is represented as an instance on the heap. This unification simplifies our extended reduction rules in Section 3.2.2 because an explicit distinction between primitive type values and addresses can be avoided. Furthermore, message send is specified as a big step operational semantics like in a publication of Clarke and Drossopoulou [CLAR 02], which makes passing of parameters and return values more explicit.

In a first step we define the syntax and operational semantics of this language (Section 3.2.1), and in a second step we define an extended language by extending the reduction rules to specify how object reference transfer is captured by aliases (Section 3.2.2). In Section 3.2.3 we proof, that the extended language preserves the semantics of the original one. This is an important property, as Object Flow Analysis like all other dynamic analyzes, should only investigate the behavior of a program but should not alter it.

3.2.1 A Minimal Object Language

We specify the dynamic behavior of a minimal object-oriented language L using a big step operational semantics. We do not define the static semantics that describes well-formed programs (*i.e.*, using a type system). From a program P we only assume sets of identifiers for class names *Class*, field names *Field*, and method names *Method*, and use variables $c \in$ *Class*, $f \in$ *Field*, and $m \in$ *Method*. Moreover, we assume that *MethodBody*(m, c) yields the expression constituting the body of the method that is returned from a lookup of m starting in class c. The function *FieldsOf*(c) returns the set of field identifiers of class c.

Syntax. In Table 3.1 we define an abstract syntax of L with source expressions *Expr*. To simplify our presentation, but without loss of generality, we define methods to have exactly one argument, referred to by the variable x. A runtime expression *RExpr* is a source expression, an address, or a call with its stack frame σ.

In contrast to usual formalizations, our runtime expressions do not have primitive type values, such as null. Consequently, all values in runtime expressions are addresses, which reference objects stored on the heap. The literal null is represented as the single object of the class *UndefinedObject* (and other literal values would be in the same way).

Dynamic aspects. The heap maps addresses ι to objects o. Objects are defined as tuples carrying their class name and a finite mapping from field names to addresses. We use the notation $H(\iota)$ to denote the lookup of the object stored at address ι on the heap H. The double lookup $H(\iota)(f)$ returns the address stored in the field f of object $H(\iota)$. The notation $o[f \mapsto \iota]$ denotes the update of o with binding $f \mapsto \iota$. Initially, the heap contains the object representing null at the fixed address 0: $H = \{0 \mapsto (\mathit{UndefinedObject})\}$.

The stack frame σ is a tuple of a target address and an argument address. We do not explicitly model the stack because it is not required to track object flow. The stack could be modeled simply by extending stack frame tuples to additionally store the caller frame.

With $\mathcal{C}$ we define evaluation contexts taking the Wright-Felleisen approach [WRIG 94].

Operational semantics We use a big step operational semantics, which is given by the relation

$$\rightarrow \subseteq (\mathit{RExpr} \times \mathit{Heap}) \times (\mathit{RExpr} \times \mathit{Heap})$$

defined in Table 3.2.

Source and runtime expressions

$$
\begin{array}{rcll}
e \in \mathit{Expr} & ::= & \mathsf{new}\ c & \textit{(new object)} \\
& | & \mathsf{this} & \textit{(this reference)} \\
& | & e.m(e) & \textit{(message send)} \\
& | & \mathsf{x} & \textit{(argument)} \\
& | & e.f & \textit{(field read)} \\
& | & e.f{=}e & \textit{(field write)} \\
& | & \mathsf{null} & \textit{(null reference)} \\
\\
e_r \in \mathit{RExpr} & ::= & e & \textit{(source expressions)} \\
& | & \iota & \textit{(address)} \\
& | & \sigma \cdot e_r & \textit{(nested call)}
\end{array}
$$

Dynamic aspects

$$
\begin{array}{rcl}
o \in \mathit{Object} & = & \mathit{Class} \times (\mathit{Field} \rightarrow \mathit{Address}) \\
\iota \in \mathit{Address} & = & \mathbb{N} \\
H \in \mathit{Heap} & = & \mathit{Address} \rightarrow \mathit{Object} \\
\sigma \in \mathit{StackFrame} & = & \mathit{Address} \times \mathit{Address}
\end{array}
$$

Reduction Contexts (call-by-value)

$$\mathcal{C} ::= [\,] \mid \mathcal{C}.m(e) \mid \iota.m(\mathcal{C}) \mid \mathcal{C}.f \mid \mathcal{C}.f = e \mid \iota.f = \mathcal{C} \mid \sigma \cdot \mathcal{C}$$

Table 3.1: Syntax and dynamic aspects.

The NEW rule creates an object with class c, and initializes all fields of the new object with null (*i.e.*, mapping fields to the address 0). The heap H is extended to the heap H' that additionally contains a binding for the new address ι.

NULL just replaces the syntactic element null with the reserved addressed 0. The heap is not modified. THIS and ARG fetch the target address, respectively the argument address, from the current stack frame. FIELD-READ looks up the object with address ι and then from this object the field f. FIELD-WRITE updates the heap with the mapping of the address ι to the object with updated mapping of the field f.

MESSAGE-SEND is slightly more complicated. First, the class c is extracted from the target object (the class is stored as the first element of an object tuple). Once c is determined, *MethodBody* performs a method lookup and returns the expression e representing the body of method. In the second step, a new call frame σ' is created as the tuple containing the target and argument. The rule says that if the expression e in the context of the call frame σ' and the heap H evaluates to the object at address ι'' with heap H', then the message send is replaced with ι'' (return value) and the new heap.

(NEW)

$$\frac{\begin{array}{c}\mathit{FieldsOf}(c) = \{f_1, ..., f_n\} \\ \iota \text{ is fresh in H} \\ H' = H[\iota \mapsto (c, f_1 \mapsto 0, ..., f_n \mapsto 0)]\end{array}}{\sigma \cdot \mathsf{new}\ c, H \rightarrow \sigma \cdot \iota, H'}$$

(NULL)

$$\frac{}{\sigma \cdot \mathsf{null}, H \rightarrow \sigma \cdot 0, H}$$

(THIS)

$$\frac{\sigma = (\iota, \_)}{\sigma \cdot \mathsf{this}, H \rightarrow \sigma \cdot \iota, H}$$

(ARG)

$$\frac{\sigma = (\_, \iota)}{\sigma \cdot \mathsf{x}, H \rightarrow \sigma \cdot \iota, H}$$

(FIELD-READ)

$$\frac{\iota' = H(\iota)(f)}{\sigma \cdot \iota.f, H \rightarrow \sigma \cdot \iota', H}$$

(FIELD-WRITE)

$$\frac{H' = H[\iota \mapsto H(\iota)[f \mapsto \iota']]}{\sigma \cdot \iota.f = \iota', H \rightarrow \sigma \cdot \iota', H'}$$

(MESSAGE-SEND)

$$\frac{\begin{array}{c}(c, \_) = H(\iota) \\ \mathit{MethodBody}(m, c) = e \\ \sigma' = (\iota, \iota') \\ \sigma' \cdot e, H \rightarrow \sigma' \cdot \iota'', H'\end{array}}{\sigma \cdot \iota.m(\iota'), H \rightarrow \sigma \cdot \iota'', H'}$$

(CONTEXT)

$$\frac{\sigma \cdot e_r, H \rightarrow \sigma \cdot e_r', H'}{\sigma \cdot \mathcal{C}[e_r], H \rightarrow \sigma \cdot \mathcal{C}[e_r'], H'}$$

Table 3.2: Reduction rules of operational semantics.

Please note that in this language there is no explicit return statement; the returned value is the value that the body of the method evaluates to.

3.2.2 The Extended Language

We now extend the language L in order to formally specify Object Flow Analysis. We introduce an additional level of indirection between runtime expressions and objects. Object addresses in the extended language L_{ext} are represented by an alias record. Conceptually, an alias in the Object Flow Analysis model represents an object reference. Similar to objects, aliases are referred to by an address and are stored, separate from the main heap, in an alias store. The syntax and basic structure of the reduction rules are kept unchanged.

Table 3.3 shows the extended syntax and dynamic aspects of L_{ext} for tracking object flow. Expressions are unchanged except for the symbol κ that we use to refer to an address of an alias. Runtime expressions do not contain addresses to objects anymore.

Source and runtime expressions

$$
\begin{array}{rcll}
e \in \mathit{Expr} & ::= & \ldots & \\
\\
e_r \in \mathit{RExpr} & ::= & e & \textit{(source expressions)} \\
& | & \kappa & \textit{(alias address)} \\
& | & \sigma \cdot e_r & \textit{(nested call)}
\end{array}
$$

Dynamic aspects

$$
\begin{array}{rcl}
o \in \mathit{Object} & = & \mathit{Class} \times (\mathit{Field} \rightarrow \mathit{Address}) \\
a \in \mathit{Alias} & = & \mathit{Address} \times \mathit{Address} \times \mathit{Address} \times \mathit{StackFrame} \\
\iota, \kappa \in \mathit{Address} & = & \mathbb{N} \\
H \in \mathit{Heap} & = & \mathit{Address} \rightarrow \mathit{Object} \\
A \in \mathit{AliasStore} & = & \mathit{Address} \rightarrow \mathit{Alias} \\
\sigma \in \mathit{StackFrame} & = & \mathit{Address} \times \mathit{Address}
\end{array}
$$

Reduction Contexts (call-by-value)

$$\mathcal{C} ::= [\,] \mid \mathcal{C}.m(e) \mid \kappa.m(\mathcal{C}) \mid \mathcal{C}.f \mid \mathcal{C}.f = e \mid \kappa.f = \mathcal{C} \mid \sigma \cdot \mathcal{C}$$

Table 3.3: Extended syntax and dynamic aspects (differences to Table 3.1 highlighted in gray).

Dynamic aspects. Addresses κ in runtime expressions refer to a binding in the alias store A, which maps addresses to aliases. An alias is a tuple $(\iota, \kappa_{orig}, \kappa_{pred}, \sigma)$, where ι is the address of the actual object, κ_{orig} is the address of the origin alias, κ_{pred} is the address of the predecessor alias, and σ is the stack frame (in Figure 3.1 referred to as MethodInvocation). Depending on the class of an alias, κ_{orig} and κ_{pred} can be undefined ($\perp$). For conciseness, the class of an alias is not stored.

We define the following convenience function that yields the object address that an alias wraps.

Definition 1 *(Object of alias)*

$$o(\kappa, A) := \iota \quad \textit{where} \quad A(\kappa) = (\iota, \_, \_, \_)$$

Intuitively, the function o takes an alias address and an alias store, looks up the alias tuple in the alias store and then yields the object address located at the first position.

(NEW)

$$\frac{\begin{array}{c}\mathit{FieldsOf}(c) = \{f_1, ..., f_n\} \\ (\kappa_1, A_1) = writeA(\sigma, A_0, 0, \bot, \bot), ..., \\ (\kappa_n, A_n) = writeA(\sigma, A_{n-1}, 0, \bot, \bot) \\ \iota \text{ is fresh in H} \\ H' = H[\iota \mapsto (c, f_1 \mapsto \kappa_1, ..., f_n \mapsto \kappa_n)] \\ (\kappa, A') = allocA(\sigma, A_n, \iota)\end{array}}{\sigma \cdot \mathsf{new}\ c, H, A_0 \rightarrow \sigma \cdot \kappa, H', A'}$$

(NULL)

$$\frac{(\kappa, A') = literalA(\sigma, A, 0)}{\sigma \cdot \mathsf{null}, H, A \rightarrow \sigma \cdot \kappa, H, A'}$$

(THIS)

$$\frac{\sigma = (\kappa, \_)}{\sigma \cdot \mathsf{this}, H, A \rightarrow \sigma \cdot \kappa, H, A}$$

(ARG)

$$\frac{\sigma = (\_, \kappa)}{\sigma \cdot \mathsf{x}, H, A \rightarrow \sigma \cdot \kappa, H, A}$$

(FIELD-READ)

$$\frac{\begin{array}{c}\kappa_{orig} = H(o(\kappa, A))(f) \\ (\kappa'', A') = readA(\sigma, A, \kappa_{orig})\end{array}}{\sigma \cdot \kappa.f, H, A \rightarrow \sigma \cdot \kappa'', H, A'}$$

(FIELD-WRITE)

$$\frac{\begin{array}{c}\iota = o(\kappa, A) \\ \kappa_{pred} = H(\iota)(f) \\ (\kappa'', A') = writeA(\sigma, A, o(\kappa', A), \kappa', \kappa_{pred}) \\ H' = H[\iota \mapsto H(\iota)[f \mapsto \kappa'']]\end{array}}{\sigma \cdot \kappa.f = \kappa', H, A \rightarrow \sigma \cdot \kappa', H', A'}$$

(MESSAGE-SEND)

$$\frac{\begin{array}{c}(c, \_) = H(o(\kappa, A)) \\ \mathit{MethodBody}(m, c) = e \\ \kappa'' \text{ is fresh in } A \\ \sigma' = (\kappa, \kappa'') \\ (\kappa'', A') = paramA(\sigma', A, \kappa') \\ \sigma' \cdot e, H, A' \rightarrow \sigma' \cdot \kappa''', H', A'' \\ (\kappa_4, A''') = \begin{cases} returnA(\sigma, A'', \kappa''') & if\ \kappa''' \neq \kappa \\ (\kappa, A'') & else \end{cases}\end{array}}{\sigma \cdot \kappa.m(\kappa'), H, A \rightarrow \sigma \cdot \kappa_4, H', A'''}$$

(CONTEXT)

$$\frac{\sigma \cdot e_r, H \rightarrow \sigma \cdot e_r', H'}{\sigma \cdot \mathcal{C}[e_r], H \rightarrow \sigma \cdot \mathcal{C}[e_r'], H'}$$

(Alias creation functions)

$$\begin{array}{rcl} allocA(\sigma, A, \iota) & := & (\kappa, A[\kappa \mapsto (\iota, \bot, \bot, \sigma)]) \\ literalA(\sigma, A, \iota) & := & (\kappa, A[\kappa \mapsto (\iota, \bot, \bot, \sigma)]) \\ paramA(\sigma, A, \kappa_{orig}) & := & (\kappa, A[\kappa \mapsto (o(\kappa_{orig}, A), \kappa_{orig}, \bot, \sigma)]) \\ returnA(\sigma, A, \kappa_{orig}) & := & (\kappa, A[\kappa \mapsto (o(\kappa_{orig}, A), \kappa_{orig}, \bot, \sigma)]) \\ readA(\sigma, A, \kappa_{orig}) & := & (\kappa, A[\kappa \mapsto (o(\kappa_{orig}, A), \kappa_{orig}, \bot, \sigma)]) \\ writeA(\sigma, A, \iota, \kappa_{orig}, \kappa_{pred}) & := & (\kappa, A[\kappa \mapsto (\iota, \kappa_{orig}, \kappa_{pred}, \sigma)]) \\ & & \text{...where} \quad \kappa\ fresh\ in\ A \end{array}$$

Table 3.4: Extended reduction rules (differences compared to Table 3.2 highlighted in gray).

Operational semantics. The runtime semantics is given by the extended relation:

$$\rightarrow \subseteq (\mathit{RExpr} \times \mathit{Heap} \times \mathit{AliasStore}) \times (\mathit{RExpr} \times \mathit{Heap} \times \mathit{AliasStore})$$

This relation is defined by the reduction rules in Table 3.4.

NEW creates two types of aliases. First, for each field in the new object, a write alias is created using the function $writeA()$ defined at the bottom of Table 3.4. The context of the write alias is the stack frame σ in which the object is created using the keyword new. Second, an allocation alias is created, which points to the new object ι and to the same stack frame σ.

NULL creates a literal alias for the object null. The rules THIS and ARG do not have to be modified, except for the address symbol ι that is replaced with κ to point out that the addresses in the stack frame point into the alias store instead of the heap.

FIELD-READ first extracts ι, the address of the actual object, which is stored at the first position of the alias tuple. Then the value of the field of the object is looked up and a read alias is created for this reference transfer. The origin alias of the field read alias is the alias κ_{orig} currently stored in the field.

FIELD-WRITE also first extracts the address of the actual object. It then looks up the current value of the field that is going to be changed. This value, aliased by κ_{pred}, is then remembered as the predecessor in the new field write alias. The origin of the field write alias is κ', the right hand side of the assignment. What this rule also shows is that the result of the assignment is the alias κ', that is, the original right hand side value rather than the newly created field write alias.

In the MESSAGE-SEND rule again two types of aliases are created. First, before the new method body is evaluated, an alias is created for the parameter. Note that the context σ' is the new stack frame, not the one in which the message m is sent. Furthermore, the target of the message send, the object address represented by the alias κ, is directly used as this in the new stack frame; no new alias is created in this case.

Under the condition that the address returned from the evaluation of e is not identical to this, a return alias is created. The rationale to not create a return alias on each return from a method call is that we want to capture only cases in which a different value than this is explicitly returned. This is especially important in languages like Smalltalk that implicitly return this if no other value is returned. In other languages the return type of such methods is usually declared void. Also important to notice with return aliases is that their context σ is the stack frame to which they are returned, rather than the one from which they originate.

Example. We illustrate the extended operational semantics by evaluating a simple example program. The example code includes a class instantiation, a method call, parameter and return value passing, and a field assignment.

The following listing shows the layout of the two classes (we use pseudo code as we have not defined a complete syntax for our language). The first class having one field, the second class having no field. Below, the main code of the example is shown. This code instantiates IRBuilder and sends the message startNewSeq. Since in our minimal language exactly one parameter is required, we use null, although in the real code the method takes no parameter.

```
class IRBuilder
    fields: currentSeq
    methods: startNewSeq(x)
class IRSequence
    fields: –
    methods: –
main { (new IRBuilder).startNewSeq(null) }
```

The following code shows the implementation of the method startNewSeq of the class IRBuilder. It creates a new IRSequence instance and assigns it to the field currentSeq of the target object.

```
method startNewSeq(x) {
    this.currentSeq := new IRSequence
}
```

In Table 3.5 the reduction steps are listed. The initial state is an empty stack frame and the source expression of the main method body. The heap is initialized with the unique instance of null, and the alias store is empty. The evaluation of the initial expression requires three reduction steps (1–3). For the MESSAGE-SEND rule of step 3, three more reductions (3.1–3.3) are required as part of the rule's premise.

(1) In the first reduction step, (NEW), the target of the messsage send is reduced, which produces the new object with address ι_1 on the heap. This address is represented by the allocation alias κ_2. The field of the new object is initialized with a write alias κ_1 pointing to null.

(2) In the second reduction step, the literal null used as method parameter, is evaluated. This step produces a literal alias κ_3 but does not modify the heap.

(3) In the third reduction step, the method startNewSeq is called. For passing the parameter κ_3, a new parameter alias κ_4 is created, which is stored together with the target κ_2 in the new stack frame. The following three reduction steps are evaluated in this new context.

$() \cdot (\mathsf{new}\ IRBuilder).startNewSeq(\mathsf{null}),$
$\{0 \mapsto (UndefinedObject)\},$
$\{\}$

$\rightarrow^{1}$ $() \cdot \kappa_2.startNewSeq(\mathsf{null}),$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_1)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..)\}$

$\rightarrow^{2}$ $() \cdot \kappa_2.startNewSeq(\kappa_3),$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_1)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..)\}$

$(\kappa_2, \kappa_4) \cdot \mathsf{this}.currentSeq := \mathsf{new}\ IRSequence,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_1)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..), \kappa_4 \mapsto (0, \kappa_3, ..)\}$

$\rightarrow^{3.1}$ $(\kappa_2, \kappa_4) \cdot \kappa_2.currentSeq := \mathsf{new}\ IRSequence,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_1)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..), \kappa_4 \mapsto (0, \kappa_3, ..)\}$

$\rightarrow^{3.2}$ $(\kappa_2, \kappa_4) \cdot \kappa_2.currentSeq := \kappa_5,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_1), \iota_2 \mapsto (IRSequence)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..), \kappa_4 \mapsto (0, \kappa_3, ..), \kappa_5 \mapsto (\iota_2, ..)\}$

$\rightarrow^{3.3}$ $(\kappa_2, \kappa_4) \cdot \kappa_5,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_6), \iota_2 \mapsto (IRSequence)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..), \kappa_4 \mapsto (0, \kappa_3, ..), \kappa_5 \mapsto (\iota_2, ..),$
$\kappa_6 \mapsto (\iota_2, \kappa_5, \kappa_1, ..)\}$

$\rightarrow^{3}$ $() \cdot \kappa_7,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_6), \iota_2 \mapsto (IRSequence)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..), \kappa_4 \mapsto (0, \kappa_3, ..), \kappa_5 \mapsto (\iota_2, ..),$
$\kappa_6 \mapsto (\iota_2, \kappa_5, \kappa_1, ..), \kappa_7 \mapsto (\iota_2, \kappa_5, ..)\}$

Table 3.5: Example evaluation in L_{ext} with extended operational semantics.

(3.1) This step simply applies the rule (THIS), which does not modify the heap or alias stack.

(3.2) The next step reduces the right hand side of the assignment, that is, it instantiates the IRSequence class, producing a new heap binding with address ι_2. In the runtime expression this object is represented by the allocation alias κ_5.

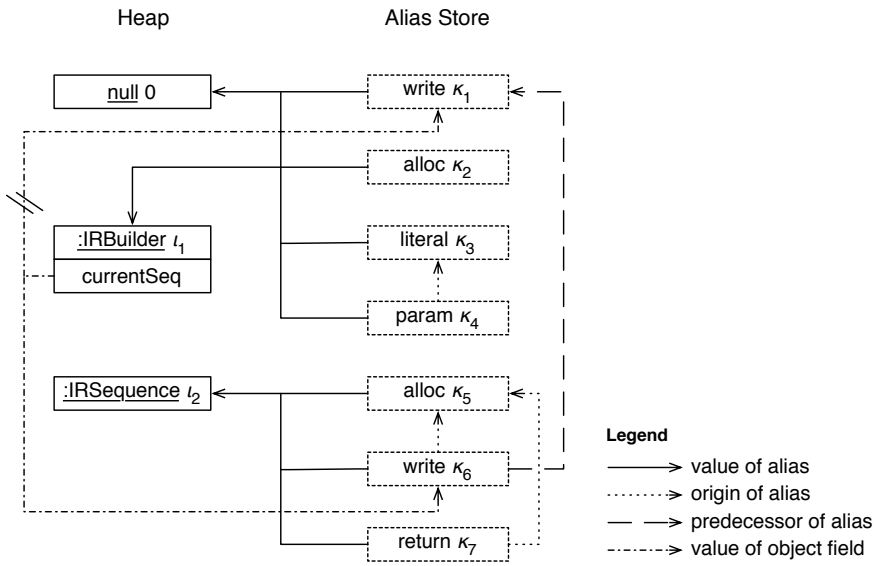

Figure 3.5: Heap and alias store of example evaluation.

(3.3) The rule (FIELD-WRITE) is evaluated, producing a write alias κ_6 as follows. First the predecessor alias κ_{pred} is obtained by looking up the current alias stored in the field. The origin alias of the field write alias is the right hand side of the assignment since the object flows from there into the field. Eventually, the field is updated with the new write alias κ_6.

Eventually, the resulting value of reduction step 3 is the return alias κ_7. In Figure 3.5 the heap and the alias store of this example evaluation are illustrated. It shows the three objects created on the heap and the 7 aliases on the alias store. The different arrows indicate references between objects and aliases. Each alias points to its object (value), and the parameter, write, and return aliases point to their origin. The write alias in addition points to the alias previously stored in the field (predecessor). The field currentSeq first points to the write alias κ_1, and later it is updated to point to the write alias κ_6.

3.2.3 Behavioral Similarity of Semantics

In the previous two sections we have defined the main language L and an extended version L_{ext}. While L defines a heap to store objects, L_{ext} additionally defines an alias store to map addresses to aliases. Aliases in L_{ext} add a level of indirection between the objects stored on the heap (field addresses are addresses of aliases instead of addresses of objects).

We wish to show that a program in L_{ext} preserves the behavior of the same program in L — that is, it merely generates additional data (aliases) but except for that the language semantics are the same. To compare the similarity of a state s of a program in L_{ext} with a state t of a program in L we define the relation F that relates s to t by flattening down the heap (and runtime expression). Figure 3.6 illustrates the flattening of the final state of the example evaluation shown in Table 3.5.

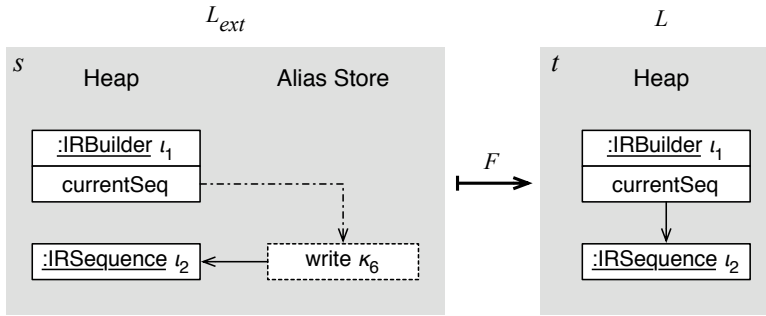

Figure 3.6: Relation F maps state s in language L_{ext} to state t in L.

Intuitively, our proof shows that at any step of the execution of a program in the extended language, its flattened heap and flattened runtime expression is identical to the heap and runtime expression of the same program being executed up to this step in the original language.

We have defined the two languages as state transition systems. L_{ext} is defined as $(S, \rightarrow)$ where $S = (RExpr \times Heap \times AliasStore)$ and L is defined as $(T, \rightarrow)$ where $T = (RExpr \times Heap)$. The definitions of $\rightarrow$ are given by the reduction rules in Table 3.4 respectively Table 3.2. We now define the *simulation preorder* F, a relation between L_{ext} and L, to show that each reduction step in L_{ext} can be matched by a step in L. The simulation we define is a *strong simulation* (or lock-step simulation) as each step in L_{ext} is matched by exactly one step in L.

Proposition 1 *The relation $F \subseteq (S \times T)$ is a simulation, that is, $(s,t) \in F$ implies that for s' with $s \rightarrow s'$ there is t' such that $t \rightarrow t'$ and $(s',t') \in F$.*

Figure 3.7 illustrates Proposition 1. Solid lines indicate hypotheses and dashed lines indicate conclusions. To verify our proposition, we first define the relation F.

The relation F flattens down a runtime expression and a heap with respect to a given alias store.

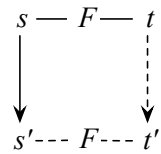

Figure 3.7: Simulation diagram

Definition 2 *(Simulation preorder F)*

$$F := \{(RExpr, H, A), (f_r(RExpr, A), f_h(H, A))\} \qquad where$$

$$f_r(e_r, A) := \begin{cases} e & if \quad e_r = e \\ o(\kappa, A) & if \quad e_r = \kappa \\ f_\sigma(\sigma, A) \cdot f_r(e'_r, A) & if \quad e_r = \sigma \cdot e'_r \end{cases}$$

$$where \qquad f_\sigma((\kappa, \kappa'), A) := (o(\kappa, A), o(\kappa', A))$$

$$and$$

$$f_h(H, A) := \{\iota_1 \mapsto (C, f_1 \mapsto o(\kappa, A), ...), ...\iota_n \mapsto ...\}$$

$$where \qquad H = \{\iota_1 \mapsto (C, f_1 \mapsto \kappa, ...), ...\iota_n \mapsto ...\}$$

Intuitively, f_r takes a runtime expression and replaces each occurrence of an alias address κ with the according object address $o(\kappa, A)$. And f_h replaces all alias addresses referred to by fields of objects on the heap with according object addresses.

Proof of Proposition 1 We case split on the reduction rules $\rightarrow$ of L_{ext}. For each rule, we first take the path to the right and then to the bottom in Figure 3.7 (that is, s mapped to t reduced to t') and then we take the other path (s reduced to s' mapped to t') to show that we obtain the identical t' along both paths.

Case (NULL): We start with this case because it is not trivial but also not the most complex one. The two listings below show each step at the left and how a step was derived at its right. What has to be proven is that both resulting states are equal.

$(\sigma \cdot \mathsf{null}, H, A)$
$(f_r(\sigma \cdot \mathsf{null}, A), f_h(H, A))$ Definition 2
$(f_\sigma(\sigma, A) \cdot \mathsf{null}, f_h(H, A))$ Definition 2
$(f_\sigma(\sigma, A) \cdot 0, f_h(H, A))$ (NULL) in L

$(\sigma \cdot \mathsf{null}, H, A)$
$(\sigma \cdot \kappa, H, A')$ where $\kappa = (0, \bot, \bot, \sigma)$ (NULL) in L_{ext}
$(f_\sigma(\sigma, A) \cdot o(\kappa, A), f_h(H, A))$ Definition 2
$(f_\sigma(\sigma, A) \cdot 0, f_h(H, A))$ Definition 1

Case (NEW): To proof this case, we first introduce the following lemma.

Lemma 1 $f_h(H[\iota \mapsto (c, f \mapsto \kappa)], A) = f_h(H, A)[\iota \mapsto (c, f \mapsto o(\kappa, A))]$
This lemma says that updating the binding of a heap in L_{ext} and then flattening this heap is equivalent to first flattening the heap and then updating the binding with the same object but with its alias addresses being unwrapped. This lemma follows directly from f_h in Definition 2.

Like in the previous case, we follow the two paths to show that they lead to the identical state:

$\sigma \cdot \mathsf{new}\ c, H, A$
$f_\sigma(\sigma, A) \cdot \mathsf{new}\ c, f_h(H, A)$ Definition 2
$f_\sigma(\sigma, A) \cdot \iota, H'$ (NEW) in L
where $H' = f_h(H, A)[\iota \mapsto (c, f_1 \mapsto 0, ..., f_n \mapsto 0)]$

$\sigma \cdot \mathsf{new}\ c, H, A$
$\sigma \cdot \kappa, H', A'$ (NEW) in L_{ext}
where $H' = H[\iota \mapsto (c, f_1 \mapsto \kappa_1, ..., f_n \mapsto \kappa_n)]$
and $A' = A[\kappa_1 \mapsto (0, ...)]...[\kappa_n \mapsto (0, ...)][\kappa \mapsto (\iota, ...)]$
$f_\sigma(\sigma, A) \cdot o(\kappa, A'), f_h(H', A')$ Definition 2
$f_\sigma(\sigma, A) \cdot \iota, f_h(H', A')$ Definition 1
$f_\sigma(\sigma, A) \cdot \iota, H''$ Lemma 1
where $H'' = f_h(H, A')[\iota \mapsto (c, f_1 \mapsto o(\kappa_1, A'), ..., f_n \mapsto o(\kappa_n, A')]$
$f_\sigma(\sigma, A) \cdot \iota, H''$ Definition 1
where $H'' = f_h(H, A')[\iota \mapsto (c, f_1 \mapsto 0, ..., f_n \mapsto 0)]$

The differences between A and A' are the new bindings for $\kappa_1, ..., \kappa_n$ and κ, which all are fresh in A, and hence $f_h(H, A) = f_h(H, A')$. Therefore, both states are equivalent.

Case (THIS): Follows directly from reduction rules and Definition 2.

Case (ARG): Follows directly from reduction rules and Definition 2.

Case (FIELD-READ): We first proof the following lemma, which is the counterpart to Lemma 1.

Lemma 2 $f_h(H, A)(\iota)(f) = o(H(\iota)(f), A)$
This lemma says that flattening a heap and then looking up a field yields the same object address as first looking up the field and then unwrapping the returned alias.

let $H = \{..., \iota \mapsto (\_, f \mapsto \kappa, ...), ...\}$ be a heap in L_{ext}, then:

$f_h(H, A)(\iota)(f)$	
$H'(\iota)(f)$	Definition 2 and H
where $H' = \{..., \iota \mapsto (\_, f \mapsto o(\kappa, A), ...), ...\}$	
$o(\kappa, A)$	field lookup in H'
$o(H(\iota)(f), A)$	binding of κ in H

Having introduced Lemma 2, we can now again proof that both paths in the case of the rule (FIELD-READ) are equivalent.

$\sigma \cdot \kappa.f, H, A$	
$f_\sigma(\sigma) \cdot o(\kappa, A).f, f_h(H, A)$	Definition 2
$f_\sigma(\sigma) \cdot f_h(H, A)(o(\kappa, A))(f), f_h(H, A)$	(FIELD-READ) in L
$f_\sigma(\sigma) \cdot o(H(o(\kappa, A))(f), A), f_h(H, A)$	Lemma 2

$\sigma \cdot \kappa.f, H, A$	
$\sigma \cdot \kappa'', H, A'$	(FIELD-READ) in L_{ext}
where $A' = A[k'' \mapsto (o(H(o(\kappa, A))(f), A), ...)]$	
$f_\sigma(\sigma) \cdot o(\kappa'', A'), f_h(H, A')$	Definition 2
$f_\sigma(\sigma) \cdot o(H(o(\kappa, A))(f), A), f_h(H, A')$	

The difference between A and A' is the update of κ'', which is fresh in A, and hence $f_h(H, A) = f_h(H, A')$. Therefore, both end states are equivalent.

Case (FIELD-WRITE): The proof of this case is analogous to (FIELD-READ), but with the difference that Lemma 1 is used instead of Lemma 2.

Case (MESSAGE-SEND): Also the proof of this rule follows the same approach like the one of the previous cases. For the intermediate reduction that is part of the rule's premise, we have to show that its left hand side is equivalent with respect to F to the left hand side of the same rule in language L. In particular, this means to show that for σ' in L_{ext} $f_\sigma(\sigma')$ is equivalent to σ' in L. This is straightforward because it requires only to show that the parameter alias created is an alias of the object passed as argument.

Case (CONTEXT): Follows directly since the rules are identical in both languages and the reduction contexts defined by $\mathcal{C}$ preserve the reduction order.

□

3.3 A Framework to Reason about Dependencies

In the introduction of this dissertation we identified *reference structure* and *reference transfer* as two main dimensions for analyzing aliasing dependencies. In this section we focus on the dimension of *reference transfer* because an

$a \in Aliases$	*(set of aliases created at runtime)*
$r \in Regions$	*(set of regions)*
$reg : Aliases \rightarrow Regions$	*(region in which an alias resides)*
$aliases : Regions \rightarrow \mathcal{P}(Aliases)$	*(set of relevant aliases of a region)*

Table 3.6: Sets and relations on which dependency definitions are based.

analysis of it can reveal dependencies not detectable with reference structure analyses. We define a small formal framework[1] to reason about dependencies introduced by the transfer of object references. This framework is then instantiated by the analyses presented in the subsequent chapters.

We have mentioned perspectives frequently taken by developers, such as objects, classes, methods, features, control flow, and threads. Between each of these program abstractions, aliasing can introduce subtle dependencies. To provide a general approach to reason about such dependencies, we abstract these perspectives with the notion of *regions*.

Regions define a partition of the set of aliases created at runtime — that is, each reference resides in a single region. Dependencies are then detected by how references are transferred between regions. The term regions is borrowed from work in static effect inference, where a region is a set of possibly aliased objects and an object is never aliased from more than one region [TALP 92]. In contrast, our concept of regions allows aliasing of an object between two regions. Indeed, this situation, which occurs when object references are transferred between regions, is exactly the focus of our analysis.

The main definitions provided by our framework are based on the set of aliases that are observed during the analysis of a program execution, referred to as $Aliases$, and the set of regions, referred to as $Regions$ (see Table 3.6). To instantiate the framework, a concrete analysis needs to define the set $Regions$ and the relation reg that defines in which region an alias resides.

Furthermore, we define the relation $aliases$, which is supposed to yield all aliases of a region that are relevant for a concrete analysis. By default, the relation $aliases$ is defined as the inverse relation of reg, but concrete analyses may override this definition to filter the aliases being considered for dependencies.

Definition 3 *(Relevant aliases of a region)*

$$aliases(r) := \{a \in Aliases : reg(a) = r\}$$

[1] We use the term *framework* in this context to denote a basic conceptual structure used to reason about a complex issue.

Based on these sets ($Aliases$ and $Regions$) and relations (reg and $aliases$) listed in Table 3.6, we define two notions of dependencies between regions. The first one defines a *direct dependency* relationship, $DD : R \rightarrow R$, the second one an *indirect dependency* relationship, $ID : R \rightarrow R$.

To simplify readability of the formulas, we use the OCL [OCL 06] notation to navigate between entities along associations defined in our metamodel in Figure 3.1. Starting from a specific object, we can navigate an association by using the opposite association-end. For example, let a be an alias, then $a.origin$ is the origin alias of a. Analogous, $a.context.method$ returns the instance of the entity Method in which the alias a is created. This notation is syntactic sugar for the functional notation, *e.g.*, $method(context(a))$.

Definition 4 *(Direct dependency)*

$$DD := \{(r, r') : \exists a \in aliases(r), reg(a.origin) = r' \wedge\ r \neq r'\}$$

Intuitively, a region r directly depends on a region r' if an object reference created in r' exists that is transferred to r without being passed through other regions in between (and the reference in r is considered relevant).

Taking not only the direct origin of an alias into account, but the whole chain of aliases from which an alias originates, we define indirect dependencies. Let $allOrigins(a) : Aliases \rightarrow \mathcal{P}(Aliases)$ be defined as the transitive closure of the $origin$ relationship of an alias a, extended to sets of aliases.

Definition 5 *(Indirect dependency)*

$$ID := \{(r, r') : \exists a \in aliases(r), \exists a' \in allOrigins(a), reg(a') = r' \wedge\ r \neq r'\}$$

Intuitively, a region r indirectly depends on a region r' if an object reference created in r' exists that is transferred to r, either directly or indirectly through other regions (and the reference in r is considered relevant).

Figure 3.8 illustrates three regions and assumes a definition of reg that maps aliases to regions as illustrated. Based on the origin relationship between aliases, for the region r we observe that it directly depends on r', and that it indirectly depends on both r' and r''.

Potentially, a variety of different meaningful instantiations of this framework exist. Its variables are (1) the set of $Regions$, that is, which entities of the model are used as regions, and (2) the definition of reg that maps aliases to these regions.

In the following three chapters we present three applications that are based on this framework. In the first one, the regions are groups of classes

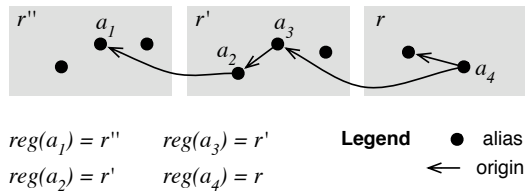

Figure 3.8: Region r directly depends on r'. Furthermore, r indirectly depends on both r' and r''.

(*e.g.*, packages), in the second one the regions are the features of a software system, and in the third one the regions are different parts of an execution trace.

3.4 Conclusion and Outlook

Object aliasing makes object-oriented programs hard to understand, maintain, and analyze. Therefore, we need a conceptual model that serves as a basis to represent and reason about object flow, which is an essential — but a so far neglected — aspect of aliasing.

In this chapter we have presented our approach, which is composed from the following three main parts as illustrated in Figure 3.9. The tracking of object flow, which we formally specify in Section 3.2, defines how object reference transfer is observed in a running system. The metamodel described in Section 3.1 defines how we represent this data, and the framework presented in Section 3.3 provides a basic conceptual structure to reason about dependencies introduced by object reference transfer.

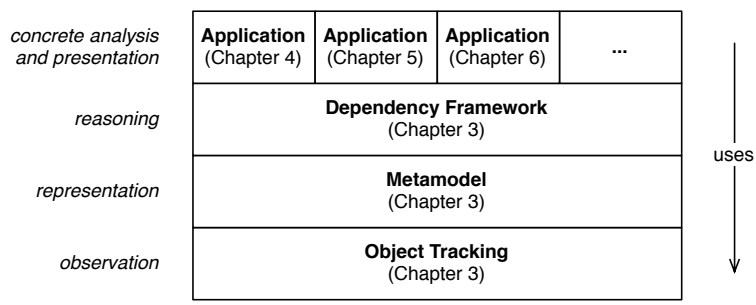

Figure 3.9: Stack diagram of the Object Flow Analysis approach.

In the following chapters we present concrete applications of our approach, which build on the abstract notion of dependencies provided by the framework. So far, we have not explicitly stated applications that can or should be carried out with Object Flow Analysis because it is not tailored to a specific set of analyses. The goal of Object Flow Analysis is to provide the foundation for a new category of dynamic analyses.

Table 3.7 maps our three applications to different views on program abstractions. In these applications we analyze dependencies between classes and packages, between features, and in execution traces. These analyses propose solutions to specific problems in the context of legacy system maintenance. Therefore, our applications cover only a fraction of the potential applications of Object Flow Analysis. The goal of these applications is to provide anecdotal evidence to validate Object Flow Analysis. They demonstrate that Object Flow Analysis provides a range of different perspectives to analyze dependencies introduced by aliasing in object-oriented systems.

Application	**Abstraction**
	Objects
Chapter 4	Static entities (classes, packages)
Chapter 5	Software features
Chapter 6	Control flow (execution traces)
	Concurrent threads

Table 3.7: Concrete applications (Chapter 4, 5, 6) mapped to abstractions between which dependencies may occur due to object reference transfer.

The first application of our approach proposes a visualization that shows how objects are passed through the classes of a system to expose how classes depend on each other by exchanging objects (Chapter 4). Our second analysis maps object flows to features to detect runtime dependencies between them based on how aliases created in one feature are subsequently used by other features (Chapter 5). Our third analysis detects how objects in execution traces are used and modified during the execution of a method to help developers write unit tests for legacy software (Chapter 6).

After presenting these three analyses, in Chapter 7 we finally propose a very different application of Object Flow Analysis that is not only based on the dimension of reference transfer but also on the dimension of reference structure. We propose the design and provide an implementation of an object-flow-aware virtual machine. The seamless integration of aliases in the memory model of virtual machines not only captures the complete execution history, but it also exhibits an unforeseen, powerful behavior: aliases no longer referenced are automatically garbage collected. Our approach proposes a solution for the long standing problem of data explosion in the field of back-in-time debugging.

Chapter 4

Visualizing Object Flow

Exposing Dependencies Between Structural Entities

While in procedural programs typically the control flow is hard to follow, in object-oriented programs the main complexity resides in the sharing and passing of objects. Domain concepts are implemented as a complex graph of objects and this graph is transformed by the program at runtime. Objects created and used in one class may later be transferred to other classes, which introduces implicit dependencies between these classes. Traditional views, like UML sequence diagrams, do not reveal the transfer of object references between classes. To solve this problem, the application of Object Flow Analysis presented in this chapter analyzes and visualizes how objects are passed through a system at runtime.

4.1 Introduction

Since the behavior of a class depends on the behavior of the classes it collaborates with, classes cannot be studied in isolation. Dependencies between classes can be difficult to detect if dependencies are implicit in the source code. While the call relationship between classes can be identified relatively well in the source code (late binding still complicates matters, though), dependencies introduced by object aliasing are less explicit. Thus, the research question that motivates the work presented in this section is:

How can we support developers to detect implicit dependencies between classes and packages that occur due to the exchange of objects?

In particular, with our approach we want to address the following concrete questions that are relevant in the context of maintaining legacy systems:

1. How do classes interact and depend on each other by exchanging objects?
2. How can we help developers to identify special classes in the design of a system (*e.g.*, classes acting as object hubs)?
3. How can we help developers to identify different phases in an execution scenario and the way classes participate in these phases?
4. Which objects passed to a class are stored in fields and hence are aliased during some period of time, and which objects are directly passed through to other classes?
5. Are internal classes of a package encapsulated or do references leak from the package?

To address these questions, we present an experimental tool that is based on Object Flow Analysis. It provides visualizations to explore the results of our analysis. We propose two explorative and complementary views to address the above stated questions:

- The *Inter-unit Flow View* depicts units connected by directed arcs subsuming all objects transferred between two units (Section 4.4). By unit we understand a class or a group of classes that a software engineer knows they conceptually belong together (*e.g.*, all classes in a package, in a component, or in an application layer, like the business logic or the user interface). This view answers the questions (1-3) stated above.
- The *Transit Flow View* allows a user to drill down into a unit to identify details of the actual objects and of the sequence of their passage (Section 4.5). This view answers the questions (4-5).

Structure of the chapter. In the next section we discuss the problem of dependencies between classes and in Section 4.3 we present the concrete dependency analysis we use to build the visualizations. In Section 4.4 and Section 4.5 we present the Inter-unit Flow View and the Transit Flow View. Section 4.6 evaluates our approach based on three case studies. Section 4.7 describes the implementation of our infrastructure that tracks objects at runtime. Section 4.8 presents the related work in program visualization. We conclude this chapter with a summary in Section 4.9.

4.2 The Challenge of Structural Dependencies

Aliasing dependencies occur when objects are passed between classes. These dependencies are particularly hard to detect when objects are passed indirectly, for example from a class A via class B to C. In this case, A and C depend on each other although they may never have exchanged messages.

These dependencies are not directly visible in the source code. Static call relationships do not expose how objects are passed around, and also UML sequence diagrams [FOWL 03], which show dynamic control flow by visualizing message sends between objects at runtime, do not expose aliasing dependencies.

The effect of such dependencies is that both classes alias the same object and hence a modification of the object by one class may affect the behavior of the other class. Even if class A discarded the reference after having passed it to B (and hence the object state cannot be mutually modified), a dependency relationship exists. The modifications of the object state as a side effect in A may affect the behavior of C at a later point in time.

Situations exist in which sharing of objects should be restricted. For example, references to instances of classes from an internal library should never be passed outside the package. This is to avoid that unknown uses of instances outside the package will break if developers modify the class assuming it has only package-internal clients. On the other hand, dependencies are not necessarily bad — often they are desired because sharing mutable state provides a high degree of expressiveness to model an application domain.

Our approach maps runtime object flow to the classes of a system. This provides a high-level view that allows developers to relate classes to each other depending on how classes exchange objects. Moreover, we allow developers to aggregate classes, for example into packages or inheritance hierarchies, to further raise the level of abstraction. This view, which exposes object flow at a high level of abstraction, can reveal insights into the design of a system.

Before introducing the visualizations, the next section discusses how to detect the dependencies, on which the view are based, using Object Flow Analysis.

4.3 Applying Object Flow Analysis

To instantiate our dependency framework introduced in Section 3.3, we define *Regions* as the set *Units*, a partition of the set of classes in the system

$a \in Aliases$	*(set of aliases created at runtime)*
$o \in Object$	*(set of objects created at runtime)*
$c \in Classes$	*(set of classes)*
$u \in Units$	*(partition of the set $Classes$)*
$r \in Regions = Units$	*(set of regions)*
$reg : Aliases \rightarrow Regions$	*(region in which an alias resides)*
$aliases : Regions \rightarrow \mathcal{P}(Aliases)$	*(set of relevant aliases of a region)*

Table 4.1: Sets and relations on which dependency definitions are based.

(see Table 4.1). That is, $Units$ is a set of nonempty subsets of $Classes$ such that every class c in $Classes$ is in exactly one of these subsets.

Furthermore, we define the relation reg as follows.

Definition 6 *(Region of alias)*

$$reg(a) := u \in Units \quad \textit{such that} \quad a.context.method.class \in u$$

Intuitively, the region of an alias is the unit that contains the class in which the alias is created (there always exists exactly one unit that contains the class because $Units$ is a partition of the set of classes). To find the class in which an alias is created we navigate along the associations context, method, and class between the entity Alias and Class in the metamodel shown in Figure 4.1.

Having provided the definitions above, we can now use the direct and indirect dependency definitions, DD and ID as defined in Section 3.3, to detect how classes depend on each other by exchanging objects. Furthermore, for our analysis, we also want to measure how many unique objects are passed along a direct dependency between two units. We define $WDD : (R \times R) \rightarrow \mathbb{N}$ as follows.

Definition 7 *(Weighted direct dependency)*

$$\begin{aligned} WDD(r, r') \quad := \quad & |\{o \in Object : \exists a \in aliases(r), \\ & a.value = o \ \wedge \ reg(a.origin) = r' \ \wedge \ r \neq r'\}| \end{aligned}$$

Intuitively, this definition counts the number of unique objects passed from class r' to class r. It does so by adapting the definition of DD. It builds a set containing objects from which an alias exists that transferred the object between the two regions, and it then yields the cardinality of this set.

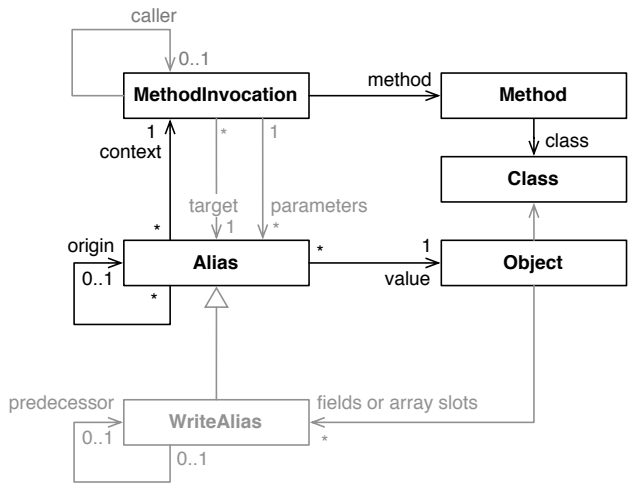

Figure 4.1: Object Flow Analysis metamodel (entities and associations used by the analysis for the proposed visualizations are highlighted in black).

4.4 Inter-unit Flow View

Figure 4.2 shows an Inter-unit Flow View produced from our Smalltalk bytecode compiler case study. As previously mentioned, the compiler has four phases: (1) scanning and parsing source code, (2) verifying and annotating the abstract syntax tree (AST), (3) translating the AST to the intermediate representation (IR), and (4) translating IR to bytecode.

The nodes in Figure 4.2 represent units (either individual classes or groups of classes), and the directed arcs represent the flows between units. An arc from a unit u' to another unit u is drawn if and only if $(u, u') \in DD$. The thickness of an arc is proportional to the number of unique objects passed along it, which is given by $WDD(u, u')$.

A force-based layout algorithm is applied (nevertheless, the user can drag nodes as he wishes). This layout results in a spatial proximity of classes and units that exchange objects.

Constructing the visualization. As the goal of our visualization is to show how objects are passed through *classes*, we aggregate the flow at the level of classes and groups of classes (units). In our experimental tool, units are specified by the developer using a declarative mapping language (similar to the approach of Walker *et al.* [WALK 98]). Rules are provided to map classes to units based on different properties such as the package they are contained

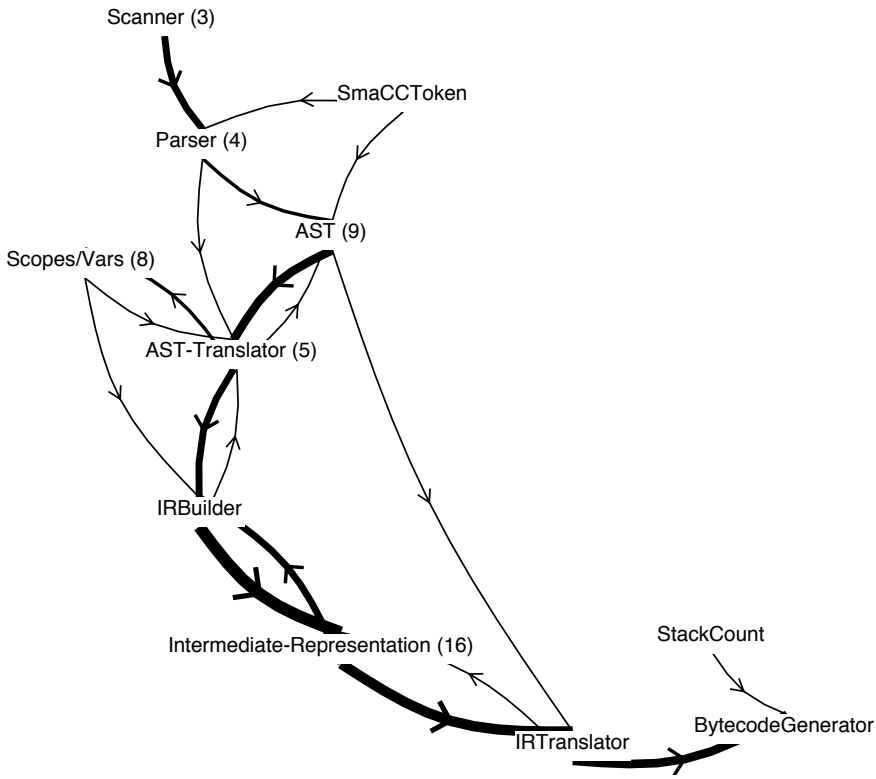

Figure 4.2: Inter-unit Flow View of the bytecode compiler.

in, their inheritance relationship, or a pattern matching their names. For instance, the first rule below maps all classes in the AST–Nodes package to the unit AST. The second rule maps IRInstruction and all classes inheriting from it to the unit IR.

```
classes containedInPackage: 'AST-Nodes' mapTo: 'AST'
classes hierarchyRootedIn: 'IRInstruction' mapTo: 'IR'
```

The following two rules gather all classes with names ending in scope or in var.

```
classes matchingName: '*scope' mapTo: 'Scopes/Vars'
classes matchingName: '*var' mapTo: 'Scopes/Vars'
```

The mapping is ordered. Each class is mapped to at most one unit, the first one for which a rule matches. If no rule matches, the class is displayed

as a single node in the visualization (*i.e.*, a unit is created that contains only this class). For convenience, there exist two additional rules:

```
classes mapAllToPackages
classes mapAllTo: 'Rest'
```

The first one maps each remaining class to the package it is contained in, that is, for each package a unit is created. The second rule maps all remaining classes to a single unit (it is syntactic sugar for matching the names with '*').

For the proposed visualization we do not take into account (i) through which instances of a class objects are passed, and (ii) the flow of objects that are only used within one class. Another important property is that we treat the flows through collections (including arrays) transparently. This means that when an object is passed from one class to a collection, and later from the collection to another class, the intermediate step through the collection is omitted in the visualization. The flow directly goes from one class to the other and there is no node created for the collection class. This abstraction makes the visualization much more concise and emphasises the conceptual flows between application classes.

Example. Let us consider again Figure 4.2, which shows the Inter-unit Flow View of the bytecode compiler case study. Various classes are aggregated to units, displayed with the number of contained classes in brackets. For instance, the group AST (9) contains the nine classes representing the abstract syntax tree.

The visualization shows which classes exchange objects. For example, there are many objects passed from the Scanner to the Parser or from Intermediate–Representation to IRTranslator. On the other hand, we also see which classes are distant in that objects only flow between them via several other classes.

Considering the thick arcs, we can detect a propagation of objects from Scanner (top) to BytecodeGenerator (bottom-right) traversing the Parser (top). This corresponds to the conceptual steps of a compiler. An interesting exceptional flow is the one from AST to IRTranslator. It contains exactly one object, the IRMethod instance that we encountered in the previous examples.

The chronological propagation of objects. The Inter-unit Flow View shows an overview of the entire execution. However, as not all objects are passed around at the same time, we are also interested in the chronological order to identify different phases of a system's execution. For example, in a program with a user interface the phases may be related directly to the exercised features.

With our tool, the user can scope the visualized object flow information to a specific time period by using a slider representing the timeline. The position of the slider defines up until which point in time object flows are taken into account. A recently active arc is displayed in dark gray which then fades and eventually becomes invisible. The goal of this feature is to help investigate how objects are propagated during a program execution.

Figure 4.3 illustrates snapshots taken after about one third, two third, and at the end of a compiling cycle (compare with Figure 4.2). In the first phase we see that objects are passed from Scanner to Parser and from Parser to AST. In the second phase, objects are mainly passed between AST and AST−Translator, IRBuilder and Intermediate−Representation. In the third and last phase, objects are transferred from Intermediate−Representation to BytecodeGenerator.

Highlighting spanning flows. With the aforementioned features we can see which units directly exchange objects and when. However, we cannot see if there exist objects that are passed from one unit to another *indirectly*, *i.e.*, spanning intermediate units.

This information is useful to understand which units act as steps in object flows leading to a unit. The same holds for the objects passed outside a unit where it is interesting to know to which other units the objects are forwarded and which paths are taken.

In our tool the user can select a unit. Thereafter, all arcs that contain objects being passed to the selected unit are highlighted in orange and all arcs with objects passed from the selected entity are highlighted in blue[1]. Remaining arcs are dashed. To determine whether objects exist that are (indirectly) passed from a unit u' to a unit u we use the indirect dependency definition (that is, $(u, u') \in ID$).

Figure 4.4 shows twice the same visualization (compare with Figure 4.2) but with different classes selected. In Figure 4.4.A Parser is selected. We see that objects are passed to it directly from Scanner (orange arc). On the other hand, the objects it passes outside reach many different units, the longest path reaches the Intermediate−Representation unit (blue arcs). In Figure 4.4.B IRBuilder is selected. We see that it depends on most above units from which it obtains objects and that it forwards objects to almost all units below.

This view highlights from where objects are passed to a unit and which routes are taken. This tells us, for example, how dependent a unit is on other units, *e.g.*, IRBuilder depends on objects created by or passed through all upper classes except for Scanner and SmaCCToken. The highlighted outgoing flows, on the other hand, tell us how influential a class is.

[1]On a B/W print, orange corresponds to light gray and blue to dark gray.

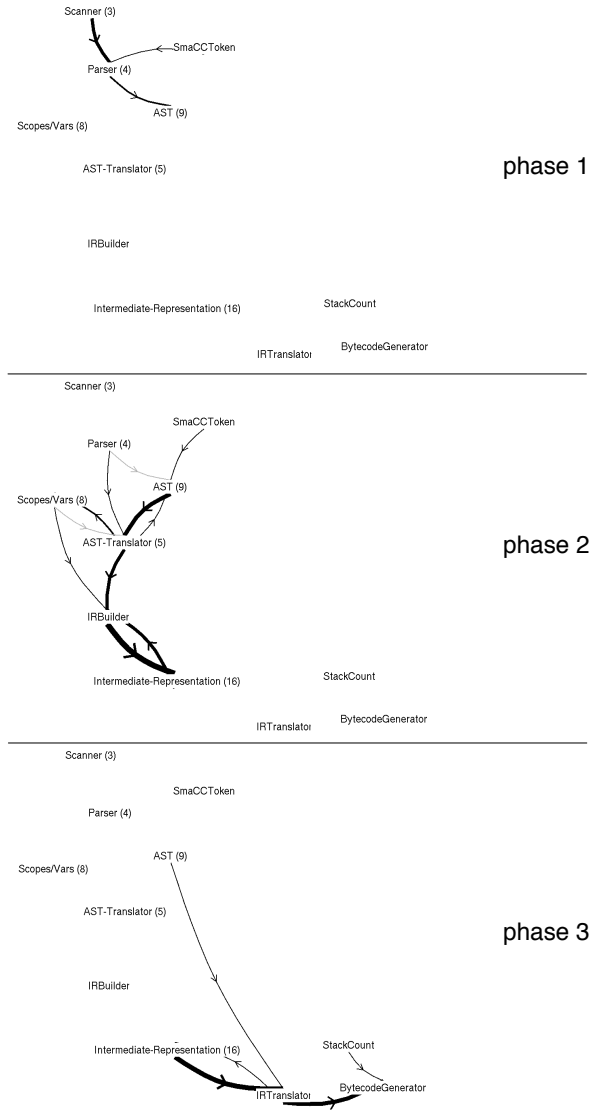

Figure 4.3: Chronological propagation of flows in the compiler.

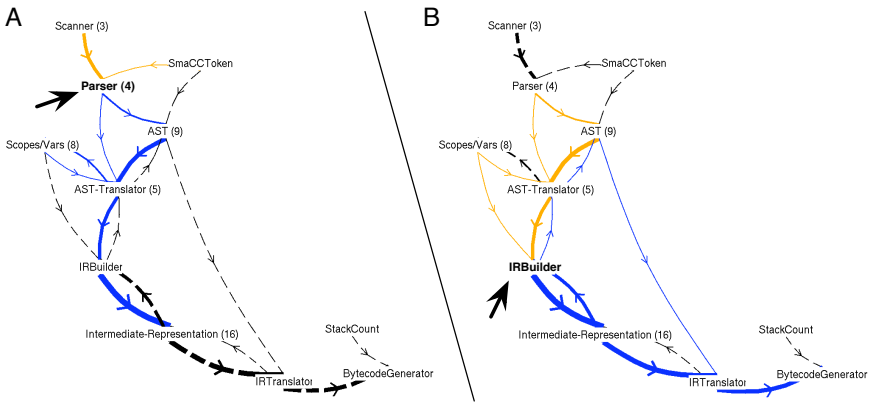

Figure 4.4: Orange and blue arcs indicate flows leading to and coming from the selected unit Parser (A), resp. unit IRBuilder (B). Dashed arcs show flows that do not contain objects coming from or leading to a selected unit.

4.5 Transit Flow View

The aforementioned visualization lacks information about the actual objects being passed through a unit. To help investigate this information, our tool allows the user to drill down to access detailed information about the objects passing through a unit.

Figure 4.5 illustrates the Transit Flow View for the class IRBuilder. It lists from top to bottom all instances that pass through IRBuilder grouped by their class. The objects inside a class are grouped by their arrival time. For each instance the point in time when it was passed into or out of the class is indicated with a rectangle. An orange rectangle shows that the object is passed *in*; a blue rectangle that it is passed *out*. A line is displayed during the time when the object is stored in a field (or contained in a collection that is stored in a field).

The Transit Flow View shows when flows take place and how many instances of which class are involved. Further exploration reveals: (1) objects passed through directly (orange/blue pairs without line), (2) objects stored in fields or collections (line), (3) objects created (the first rectangle is not orange, therefore, the object is created in the class), and (4) objects passed in or out multiple times (several rectangles for the same object).

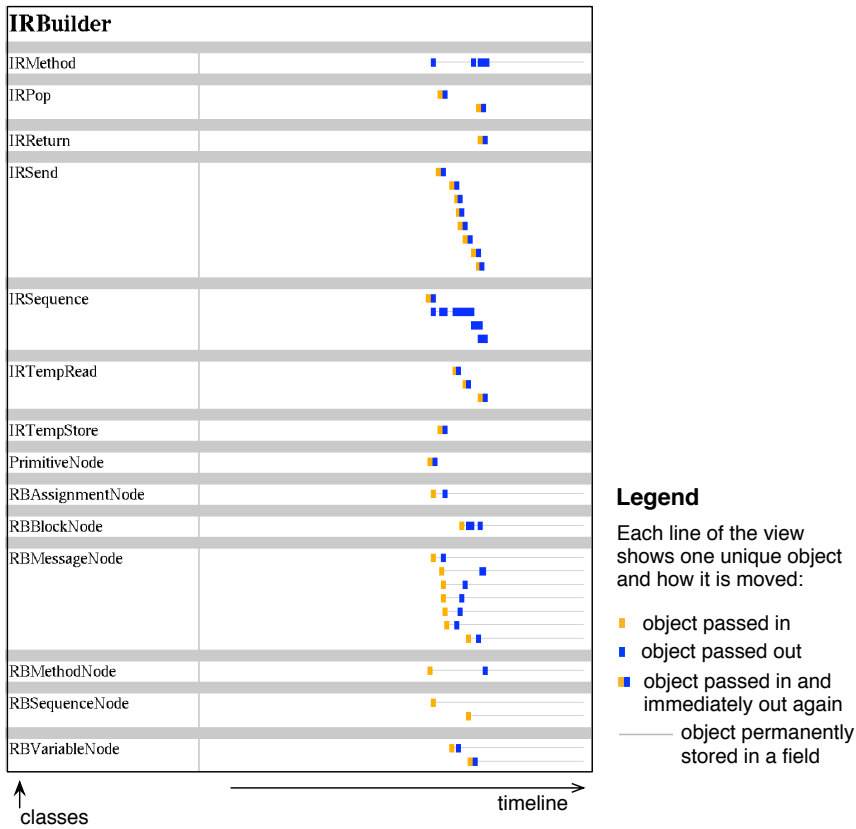

Figure 4.5: IRBuilder Transit Flow View.

For example, in Figure 4.5, the IRMethod instance is created in IRBuilder, it is stored in a field, and it is passed out multiple times. Intermediate representation instances (classes with names starting with IR) are passed through IRBuilder, whereas the instances at the bottom (AST nodes) are also stored.

4.6 Case Studies

In this section we provide an overview of the results we obtained from applying the visualizations to three case studies: a Smalltalk bytecode compiler, a health insurance web application and an IRC chat client. All three applications are implemented in Squeak [INGA 97], an open-source Smalltalk dialect [GOLD 83]. Our choice of those case studies was motivated

by the following reasons: (1) they are non-trivial and model very different domains, (2) we have access to the source code, and (3) for the compiler and health insurance application we have direct access to developer knowledge to verify our findings.

The objective of these preliminary investigations is to evaluate the usefulness of the two visualizations of our approach and to learn about a practical exploration process using our tool.

4.6.1 Bytecode Compiler

To generate experimental data we run the compiler on a typical method source code, which includes class instantiations, local variable usage, a conditional and a return statement.

The Inter-unit Flow View illustrated in Figure 4.2 shows the final state of the view after several iterations of exploring and refining the mappings of units. Using the Inter-unit Flow View we could extract the key phases of the compiler. This was straightforward from studying the chronological propagation of the object flows. The activity starts on top (Scanner and Parser) and then shifts downwards to center around AST−Translator and IRBuilder and eventually shifts to IRTranslator and BytecodeGenerator (see Figure 4.3). This observation is in line with the documentation, which describes the following main phases: (i) scanning and parsing, (ii) translating AST to the IR, and (iii) translating the IR to bytecode.

With the help of the highlighting feature we obtained more detailed knowledge about the system. For example, IRBuilder plays a key role as it is a hub through which objects from the upper units in the view are passed to the lower ones. Using the Transit Flow View (see Figure 4.5) we studied detailed interrelationships between the units. For example, in the transition from AST to IR (phase 2) we see that the unit AST−Translator passes AST nodes (classes with the RB prefix) to IRBuilder. In Figure 4.5 we see that AST nodes are passed into IRBuilder and from the Figure 4.2 we see that they come from AST−Translator. In the Figure 4.5 we also see that IRBuilder creates three sequence objects which are passed outside multiple times.

Surprisingly, also the AST nodes are forwarded to the Intermediate Representation package (we expected that after IRBuilder created the IR from the AST, the AST nodes are discarded). Following the flow of the AST nodes we reach the Intermediate Representation package. Figure 4.6 shows the Transit Flow View of the IR package. We see here that the AST nodes are passed in and are then stored in the class but are never passed out again. This points to the fact that IR objects hold a reference to the AST node from which they originate. Also interesting in Figure 4.6 is that one can distinguish two phases of activity. The first phase is when the IR is built, where we see IR and AST nodes being passed into the IR package (marked

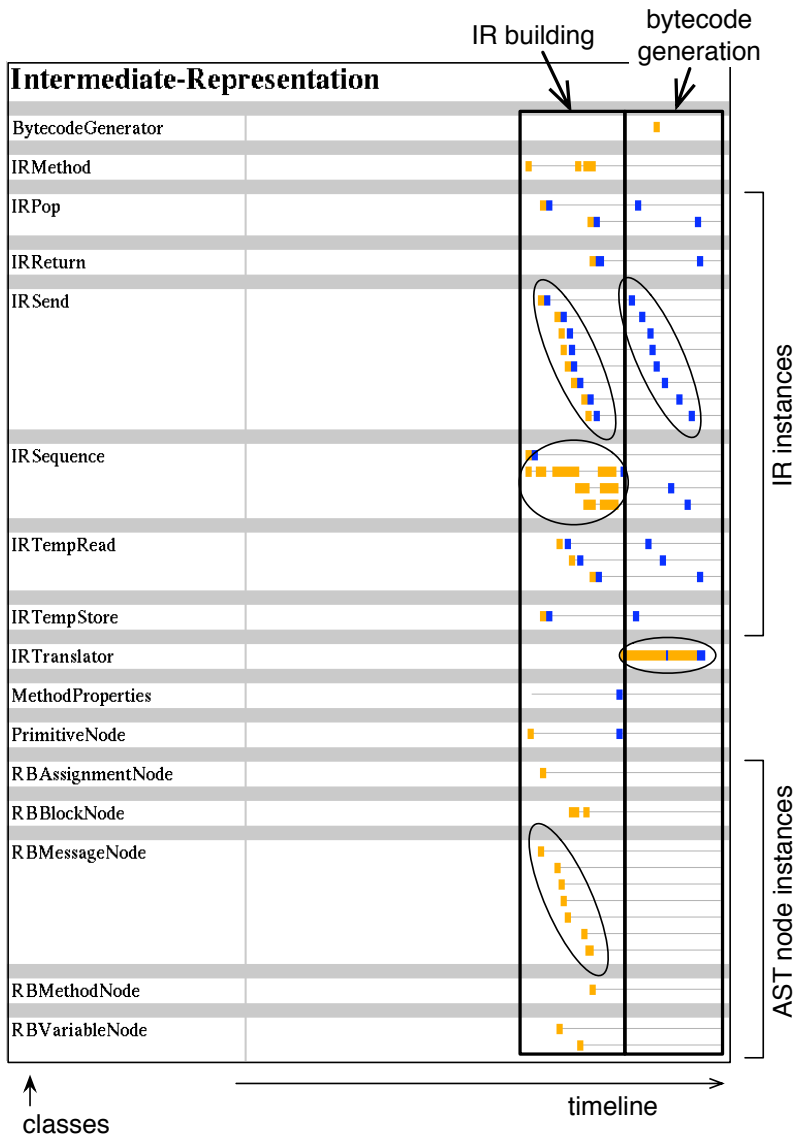

Figure 4.6: Transit Flow View of the Intermediate Representation package.

in Figure 4.6). The second phase is when the bytecode is generated, where IR objects are passed outside (but not AST nodes). In this phase we can also see that the single instance of IRTranslator is transferred many times.

In the the remaining part of this section we want to shed light on an interesting aspect of our approach we noticed in this case study.

Inversion of control flow. There are two ways how objects are passed to an instance: (i) objects are pushed to an object by being passed as method arguments, or (ii) objects are pulled by the instance by being passed as return value in response to a message send. In the latter case (ii) the objects flow in the opposite direction compared to message sends. Therefore, the object flows do not necessarily evolve in the same direction as the control flow.

For instance, the Parser creates the Scanner and then regularly accesses it to get the next token. An analysis of the execution trace shows the call relation Parser → Scanner. The object flow view, on the other hand, shows the conceptually more meaningful direction Scanner → Parser. The reason is that with Object Flow Analysis we can provide object-centric views, which abstract implementation details, such as the distinction of sender and receiver of a message. This trait also clearly distinguishes our approach from the ones that are based on dynamic control flow analysis in which method call edges point from the sender to the receiver class of a method invocation [ZAID 05, DE P 94].

4.6.2 Insurance Web Application

This industrial application was put into production six years ago and since that time has undergone various adaptations and extensions. The analyzed scenario comprises the oldest and most valuable part for the customer, the process of creating a new offer. It is composed of 10 features, including adding persons, specifying entry dates, selecting and configuring products, computing prices, and generating PDFs.

With this case study we focus our discussion on the exploration process, rather than on the details of the actual findings.

Step 1: Creating coarse-grained units. We started by investigating the Inter-unit Flow View with units corresponding to packages. However, the view was hard to work with because it was cluttered with many small packages that were part of the GUI layer (see Figure 4.7).

Step 2: Re-grouping to appropriate units. As a first refinement of the mapping of classes to units we put all classes of web GUI related packages into one unit, representing the presentation layer.

```
self containedInPackage: 'PLWeb*' mapTo: 'Web app UI layer'
```

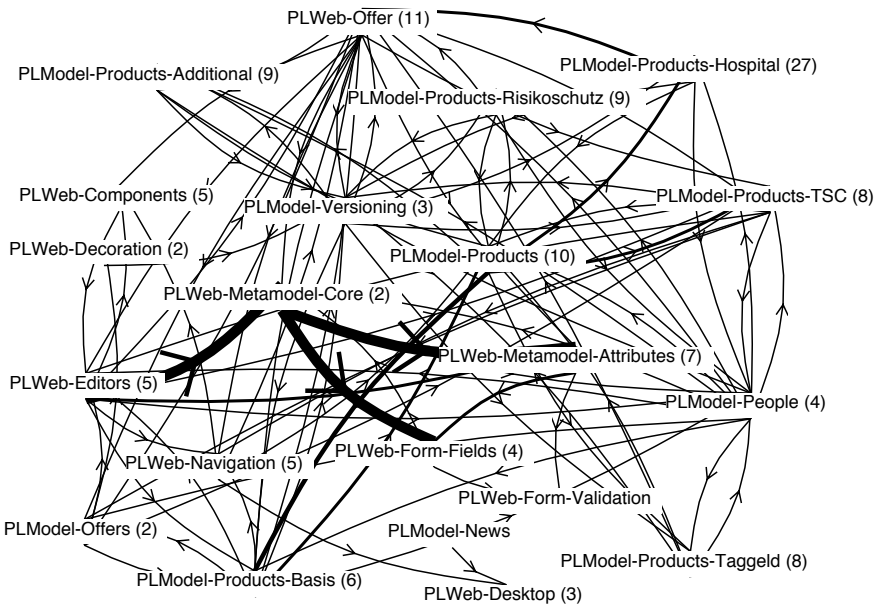

Figure 4.7: Inter-unit Flow View of the insurance application case study *before* refining the mapping. In this state it is too cluttered for comprehension.

The resulting view was already more concise. Now focusing on the business logic, we saw many packages corresponding to individual products, each package containing the product classes and associated calculation model classes. We re-grouped the classes into a Products unit and a Calculation Models unit because we wanted to learn about the higher-level concepts rather than how single products differ.

```
self hierarchyRootedIn: 'PLProduct' mapTo: 'Products'
self hierarchyRootedIn: 'PLCalculationModel' mapTo: 'Models'
```

This change dramatically improved the view. We obtained only nine units and we could identify interesting flows between them (see Figure 4.8). For instance, with the help of the Transit Flow View, we could understand how versioning works and how products and calculation models relate to each other. Products pass dates to the package responsible for versioning and in turn calculation models are passed to the products (we used the Transit Flow View to access this information).

Step 3: Extracting interesting candidate classes. Once we gained an overview, we started to dig deeper. By refining the mapping rules we

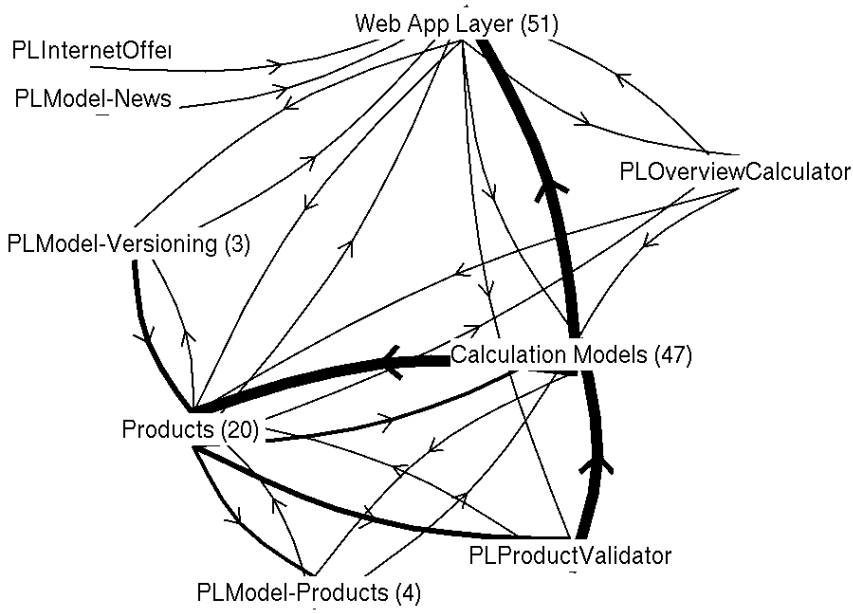

Figure 4.8: Inter-unit Flow View of the insurance application case study.

split off packages to show individual classes, *e.g.*, ProductValidator which is packaged in PLModel–Products, but from its name does not seem to be a product but rather provides specific behavior. Another similar candidate class is OveviewCalculator.

To obtain more details of those classes we used the highlighting feature to show which other units are involved in passing objects with respect to the selected class. In the case of OverviewCalculator we see that this class passes persons and dates to products, which eventually return price objects.

Discussion. The exploration process we took, which proved useful, was to first gain a coarse-grained view (step 1), find appropriate units (step 2), and only then get into more detail (step 3). Our internal declarative mapping language was helpful to create conceptual groups of classes with varying level of detail. It was essential to be able to structure units differently compared to packages. For instance, classes representing products and classes representing calculation models were organized together in the same packages. However, we wanted to distinguish products and their calculation models and hence created two units, one for all classes inheriting from Product, and the other for all classes inheriting from CalculationModel.

From the Inter-unit Flow View, conceptual relationships between units were intuitively understandable. The presented information is high-level and thus appropriate for studying the high-level design of an unfamiliar system. Yet, means are provided to drill down to gain more detailed knowledge where appropriate.

In contrast to the compiler case study, the feature for investigating the chronological propagation of objects was not particularly useful. A plausible explanation is that the compiler has a much stronger notion of sequentially transforming one representation to another. The exercised features of the health insurance application, on the other hand, do not exhibit this characteristic.

4.6.3 IRC Chat Client

As the last case study we chose an IRC Chat Client. A total of six developers contributed to this open source project, which underwent various refactorings and enhancements over nine years. We analyzed nine features: open, setup, connect to server, request MOTD, join channel, send and receive message, opening new console, and disconnect.

In contrast to the other two case studies, we enhanced the Inter-unit Flow View with information about features. We wanted to study if two classes exchange objects in only one feature or in several features.

Figure 4.9 illustrates the Inter-unit Flow View after grouping all GUI classes into one unit. The largest flows are going into and out of IRCConnection. These flows take place in several features (the darkest gray corresponds to 5 features, and the flow between IRCConnection and IRCChannelObserver takes place in 2 features). On the other hand, many flows around the GUI unit are specific to one feature only. For example IRCConnectionProfile only passes objects to the GUI classes during the *setup* feature. Another example is the class IRCChannelInfo, where flows either take place in the feature *join channel* or in the feature *disconnect* — even though the edges are relatively thick (20 objects being transferred along the thickest edge).

4.7 Implementation

For this analysis of object flow, we implemented the dynamic analysis technique in Squeak [INGA 97], an open-source Smalltalk dialect [GOLD 83], and the metamodel in Visual Works Smalltalk using the reengineering platform Moose [NIER 05] and the visualization engine Mondrian [MEYE 06].

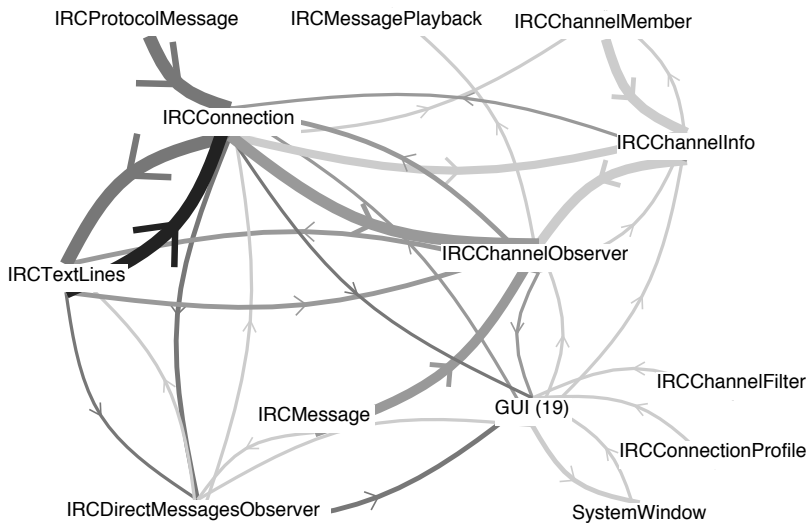

Figure 4.9: Inter-unit Flow View of the IRC chat client case study with gray toning of edges indicating the number of participating features.

To cover all object flows in a program execution as specified by Object Flow Analysis, the dynamic analysis technique has to be implemented carefully. Our Object Flow tracer not only tracks objects of application classes but also instances of system classes and primitive type values. For example, collections and arrays have to be taken into account as they preserve permanent object references between the holder of the collection and its contained elements.

In this implementation, each regular object reference is substituted by an alias object that behaves like a proxy in that it forwards messages to the real object. Every time a reference is transferred, we create a new alias instance. The hooks in a method to invoke the code that creates aliases are introduced through the bytecode instrumentation framework Reflectivity [DENK 08], and to capture argument passing and return values, we use the technique of method wrappers [BRAN 98].

The recursion problem, which typically occurs if code is instrumented and this code is also used by the tracer, is solved by applying the Twin Class Hierarchy approach [FACT 04]. The underlying idea is to create a temporal, isolated copy of all instrumented classes that is used by the application to be analyzed. The application under investigation is then executed using this modified version of the system, whereas the invoked behavior of the tracer uses the original standard system classes.

Limitations. This approach to object flow tracking is implemented completely at the application level as it uses bytecode instrumentation and some reflective capabilities of the system, but it does not require any changes to the virtual machine. While this is an advantage as the implementation is more straightforward, it has a few limitations. The main problem is that proxy objects (aliases) cannot be made completely transparent. Because the virtual machine does not know anything about aliases, some operations like equality comparison or retrieving the class of an object are applied to the alias instance instead of the actual object. In our implementation, we managed to work around most of those cases by changing the compiler to generate different bytecode in those cases.

4.8 Related Work in Program Visualization

Visualizations of dynamic control flow. Many different approaches exist to visualize dynamic control flow in object-oriented systems. For instance, Lange and Yuichi built the Program Explorer to identify design patterns by visualizing message passing between objects and classes [LANG 95]. Pauw *et al.* propose a tool to visually present execution traces to the user [DE P 98]. This approach automatically identifies reoccurring execution patterns to detect domain concepts that appear at different locations in the method execution trace. Jerding *et al.* propose ISVis, a tool to visualize interactions in program execution to help understanding the architecture of a program [JERD 97]. Walker *et al.* propose high-level views by analyzing calls between instances [WALK 98]. Similar to our approach, ISVis and the approach by Walker *et al.* allow the user to group related entities to rise the abstraction from the low-level behavior to the architectural level.

While the above mentioned approaches target understanding a system from the perspective of the flow of control, our approach provides a complementary view based on the the flow of objects. None of the above mentioned approaches captures how objects are propagated through a system at runtime.

Visualizations based on static analyses. A large body of research has been conducted into facilitating program comprehension through static analysis. Typically, such static analyses are based on a data dependence graph in which edges represent the flow of data between actual parameters and formal parameters of procedures. The data dependence graph resembles our object flow trees projected on the static control flow graph. Therefore, similar visualizations like we present in this chapter may also be produced from a static data flow analysis. In comparison to our dynamic

analysis, the static analysis would yield significantly more different object flow paths. Balmas and Krinke have independently worked on visualizations of data and control flow information, facing the problem that those graphs become huge already for small programs [BALM 01, KRIN 04]. The reason is that a static analysis provides a conservative view, which in some cases may even include infeasible execution paths of the program.

In contrast, our dynamic analysis produces a precise under-approximation. The application presented in this chapter attacks the difficulty that dynamic analysis produces large amounts of data by aggregating the flows between classes and only showing details when the user drills down using the Transit Flow View. Furthermore, our dynamic analysis approach allows the user to constrict the analysis to the software features of interest and thus he can directly relate the selected feature to the obtained results. Our approach trades off precision for completeness, and therefore we have to anticipate that the results do not apply to all possible program executions.

Combined analyses. To provide more precise views, static analysis has been combined with dynamic analysis. Quante *et al.* use execution traces to slice the control flow graph with respect to the runtime usage of a selected object [QUAN 06].

The resulting Object Process Graphs are much more concise and hence can be visualized. Object Process Graphs are similar to our visual approach in that the usage of objects is projected on the static program structure. In contrast to Object Process Graphs, our approach reveals the continuous flow of objects whereas the approach of Quante *et al.* reveals the continuous flow of control. Their approach reveals how control flow progresses between different locations in which a selected object is used. Control flow, however, does not reveal the dependencies introduced by object aliasing.

4.9 Summary of the Chapter

A key characteristics of object orientation is the deep collaboration of objects to accomplish a complex task. Understanding such applications is then difficult since reading the implementation of the classes only reveals the static aspects of the computation. At runtime, however, dependencies between structural software entities may occur that are not explicit in the source code.

In this chapter we propose an approach to visualize the flow of objects between classes and groups of classes (*e.g.*, packages) to expose these dependencies to developers. We apply Object Flow Analysis and the conceptual

framework, which is based on our metamodel. Simply by providing a definition for the relation that maps aliases to regions (in this application a region is a set of classes), we obtain the desired notion of dependency through the provided direct and indirect dependencies definitions. One additional definition is required to compute the weight of dependency arcs.

Chapter 5

Feature Dependencies

Exposing Indirect Dependencies Between Features

The domain-specific ontology of a software system includes a set of features and their relationships. While the problem of locating features in object-oriented programs has been widely studied, runtime dependencies between features are less well understood. Features cannot be understood in isolation, since their behavior often depends on objects created and aliased in previously exercised features. It is difficult to spot runtime dependencies between features just by browsing source code. Hence, code modifications intended for one feature, often inadvertently affect other features. In this chapter, we propose an approach to precisely identify dependencies between features based on Object Flow Analysis. The results of two case studies indicate that our approach helps software maintainers in understanding critical feature dependencies.

5.1 Introduction

A feature is a unit of domain functionality as understood from the user's perspective. During requirements analysis, relationships between features are specified to express conceptual dependencies and constraints of a system [RIEB 03]. Correct specification of dependencies is vital to ensure correct behavior of a system and to avoid behavioral problems.

Much of the feature-related research for system comprehension focuses on *feature identification*, a technique for locating parts of code that

implement features [WILD 95, WONG 00, EISE 05b, ANTO 05]. Only few researchers have investigated relationships between features [SALA 04, GREE 05, KOTH 06].

The runtime behavior of object-oriented systems is characterized by objects and message sends. Objects may be long-lived and used by many different features of a system. Before a feature can be exercised, it may require other features to establish a particular program state. Relationships between features can be defined based on shared usage of static entities like classes and methods [GREE 05]. A static perspective, however, overlooks runtime characteristics of object-oriented systems. We consider a runtime dependency to exist between features if state changes in one feature impact the behavior of another feature.

Thus, the underlying research question of the work we present in this chapter is: *How can we support developers to discover hidden dependencies between features to reduce the risk that modifications intended for one feature inadvertently break seemingly unrelated features?*

Salah *et al.* described a technique to identify runtime dependencies between features by detecting situations where objects are created in one feature and are later used in another feature [SALA 04]. However, considering only object instantiation is not sufficient to detect all runtime dependencies — we also need to consider *object aliasing*. The approach of Salah *et al.* only considers object creation, thus it misses dependencies between features that result from one feature accessing an object through a reference established in a feature other than the one where the object was originally instantiated.

We propose to apply Object Flow Analysis for a more precise detection of runtime dependencies between features, and we propose a visualization to expose the detected dependencies in a way they are useful for developers. First, we discuss the challenge of dependencies introduced by object aliasing and how Object Flow Analysis can detect these dependencies. Second, we present our solution to make the dependencies in object graphs explicit for the developer.

The latter is important because merely detecting which features depend on each other is not sufficient to support software maintenance. Object aliasing introduces dependencies because the object graph is persisted between exercising features and each feature modifies this graph. Features may then use existing object references and hence their behavior depends on previously exercised features that have modified these references. Therefore, to support the developer, we have to make these dependencies in an object graph explicit. By making the dependencies explicit, the developer can draw connections between the code he is modifying and other parts of the system that depend on it (or vice versa).

Chapter structure. We start this chapter by identifying that current approaches miss indirect feature dependencies caused by object aliasing (Section 5.2). In Section 5.3 we present a dependency detection strategy based on Object Flow Analysis that addresses the identified shortcoming, and in Section 5.4 we present a visual approach that allows the developer to explore these dependencies. In Section 5.5 we apply our approach to two case studies and detail the results we obtained. Section 5.6 presents related work in the field of feature analysis, and Section 5.7 concludes the chapter.

5.2 The Challenge of Feature Dependencies

Functional requirements are often centered around features since they reflect the end-user's perspective of a system. We adopt the definition of a feature proposed by Eisenbarth *et al.*: *"A feature is a realized functional requirement of a system. A feature is an observable unit of behavior of a system triggered by the user"* [EISE 03].

5.2.1 Runtime Dependencies Between Features

The behavior of one feature may depend on certain program state being established during the exercising of another feature. For example, in a Mail Client application, a "send mail" feature may require a "compose mail" feature to set mail recipients before it can be exercised.

During program execution, the object reference graph steadily changes, as new objects are created, references between objects are changed, or objects are garbage collected. As a feature is exercised, it typically produces side effects. Therefore, since program behavior depends on the reference relationships of objects, the behavior of a feature may be influenced by a previously exercised feature. To analyze these dependency situations we have to consider the interrelationships between objects. The analysis and understanding of these interrelationships is complicated by the fact that there may exist multiple access paths to the same object due to object aliasing.

5.2.2 Why Object Aliases Cause Dependencies

Intuitively, we consider a feature to depend on another one if object state is changed in the first feature and then the second feature's behavior uses this state.

Salah *et al.* [SALA 04] define a relationship $depends$ from a feature F_i to a feature F_j if F_i uses objects that are created by F_j (*i.e.*, F_i *depends on* F_j). Let $I(F)$ be the set of objects used by F (*i.e.*, the objects imported by a

feature), and $E(F)$ be the set of objects created by F (*i.e.*, the objects that can be exported by a feature), then:

$$depends \equiv \{(F_i, F_j) \mid I(F_i) \cap E(F_j) \neq \emptyset, i \neq j\}$$

To *use* an object in this context means that the object is sent a message (not including changing references to it). This definition, however, does not capture all runtime dependencies. Let us consider the following simplified case illustrated by Figure 5.1. It shows the features *Startup*, *Join Channel*, and *Receive Message* of an IRC chat client from one of our case studies.

Between each feature, we show the snapshot of live objects; the first taken before running the *Join Channel* feature and the second before the *Receive Message* feature. We treat a snapshot as a directed graph, commonly termed *object reference graph*. Nodes represent objects and edges represent a field of one object referring to another object.

While exercising the feature *Startup*, two objects, a window object (w) and connection object (c), are created. Then, while exercising the feature *Join Channel*, the first snapshot is transformed into the second. It creates an observer object (o) and assigns the connection object to one of its fields, *i.e.*, a reference $o \rightarrow c$ is created. Now the object c is aliased since there are two objects referring to it.

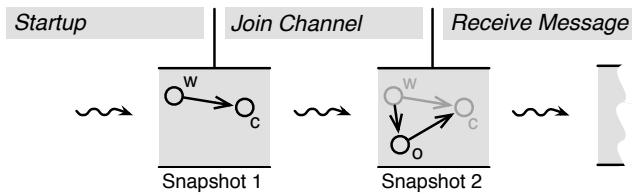

Figure 5.1: State changes between features.

Let us assume that in the *Receive Message* feature, the observer sends messages to the connection through the object reference $o \rightarrow c$. Therefore, the *Receive Message* feature depends on the *Join Channel* feature. The rationale is that without the appropriate state changes in *Join Channel*, *Receive Message* would exhibit a different behavior, or in the worst case abort with a null pointer exception.

The $depends$ relationship proposed by Salah *et al.* [SALA 04] is not capable of detecting that *Receive Message* depends on *Join Channel*. It only detects the dependency on the *Startup* feature, in which the connection is instantiated.

We conclude that a dependency detection strategy as described by *depends* at the object level (*i.e.*, capturing object creation events) is not precise enough to detect this type of indirect dependencies.

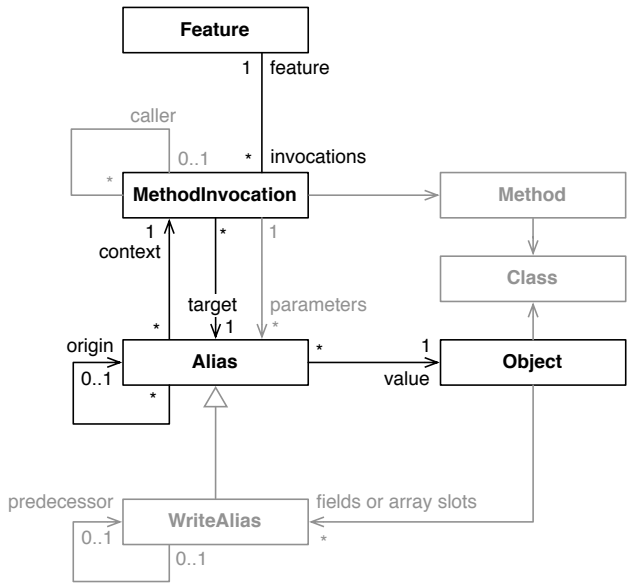

Figure 5.2: Object flow metamodel extended with Feature (entities and associations exercised by the feature dependency analysis are highlighted in black).

5.3 Applying Object Flow Analysis

We now define a new detection strategy for fine-grained dependencies based on Object Flow Analysis. For this analysis we integrate the notion of features into our metamodel as proposed by Greevy [GREE 07]. Our metamodel with the entity Feature is illustrated in Figure 5.2. Features partition the set of method invocations; each method invocation belongs to exactly one feature. During an analysis run of a system, the developer manually marks the start and the end of each exercised feature.

Similar to the previous application, which detects dependencies between classes, this application uses the dependency framework to reason about feature dependencies. As shown in Table 5.1, we define the set $Regions$ as the set of features exercised during an execution of the system, and we define the relation reg as follows.

Definition 8 *(Region of alias)*

$$reg(a) := a.context.feature$$

$a \in Aliases$	*(set of aliases created at runtime)*
$f \in Features$	*(set of features exercised at runtime)*
$r \in Regions = Features$	*(set of regions)*
$reg : Aliases \rightarrow Regions$	*(region in which an alias resides)*
$aliases : Regions \rightarrow \mathcal{P}(Aliases)$	*(set of relevant aliases of a region)*

Table 5.1: Sets and relations of the new feature dependency definition.

Intuitively, the region of an alias is the feature being exercised at the time when the alias is created.

Furthermore, like in the definition of $depends$ given in Section 5.2.2, we want to take only aliases into account to which messages have been sent in a feature — that is, objects merely passed around while exercising a feature do not trigger a dependency. This can be expressed in our framework by redefining $aliases(r)$ to return only a subset of the aliases created in a region r.

Definition 9 *(Relevant aliases of a region; redefinition of Definition 3):*

$$aliases(r) := \{a \in Aliases : \exists i \in r.invocations,\ i.target = a\}$$

Having provided the definitions above, we can now use the indirect dependency definitions, ID (Definition 5 from Section 3.3), to define the new runtime dependency relationship $depends_{new}(f, f')$ to show that f depends on f'.

$$depends_{new} \equiv ID$$

For convenience, we again show the definition of ID.

$$ID := \{(f, f') : \exists a \in aliases(f), \exists a' \in allOrigins(a), reg(a') = f' \wedge\ f \neq f'\}$$

Intuitively, a feature f depends on another feature f' if any object the feature f sends messages to can be traced back as originating from f'.

This definition represents our detection strategy and yields a superset of dependencies compared to $depends$ as defined in Section 5.2.2. For each object created in a feature, there exists an alias that is the root of all subsequently created aliases. Therefore, for each object $depends_{new}$ includes the feature in which the object was created.

Our new definition detects additional dependencies caused by object references created in features other than the one in which the object was instantiated.

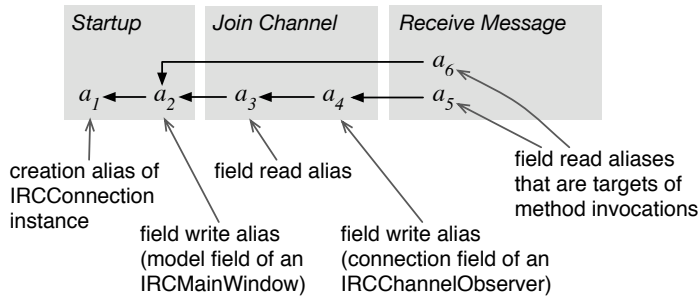

Figure 5.3: Object flow of an IRCConnection instance.

Example. Figure 5.3 illustrates an example from our IRC chat client case study. It shows the aliases of an IRCConnection instance created in the features *Startup*, *Join Channel*, and *Receive Message*. In the feature *Receive Message*, the connection instance is used as the target of method invocations in two different objects. The first alias (a_5) is created when the field named connection is read (in a channel observer instance), and the second alias (a_6) is created when the field named model is read (in a window instance).

To determine whether the *Receive Message* feature has a dependency on other features with respect to the connection instance, we backtrack the flow of this object.

On the one hand, the field read alias a_5 first leads back to *Join Channel*, and further back it leads to *Startup*. On the other hand, the origin of the field read alias a_6 is the field write alias a_2 that is created in the feature *Startup*. Based on the definition of $depends_{new}$, we conclude that the *Receive Message* feature depends both on the *Join Channel* and on *Startup* features.

Using the definition $depends$, however, we would not be able to detect the dependency on *Join Channel*, because the object we track is not created in this feature. Yet, *Join Channel* influences the behavior of *Receiver Message* by aliasing the connection instance.

In the remainder of this section, we discuss how we support a software engineer to understand feature dependencies by providing information about how the dependencies are related to each other.

5.4 Exposing Dependencies in Object Graphs

With our detection strategy, we obtain for a feature under investigation, a set of other features it depends on. Considering single object references that contribute to a dependency by being transferred between two features, a

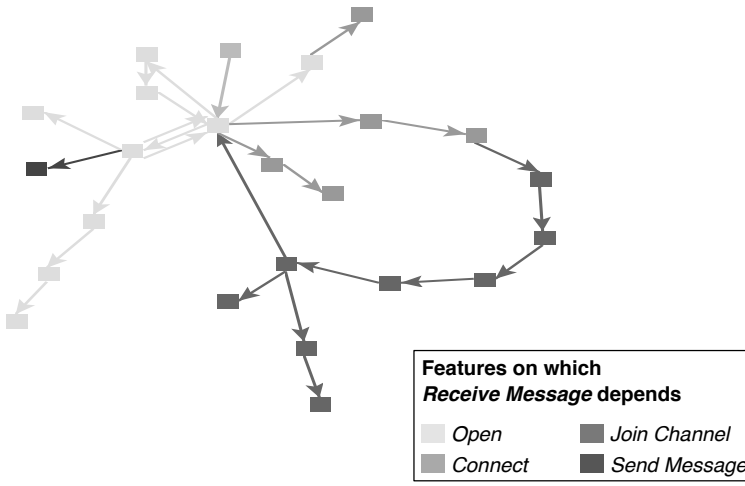

Figure 5.4: Object dependency graph of the *Receive Message* feature.

developer may have to browse a huge amount of data. Yet, investigating these datapoints is required when carrying out maintenance work to be able to draw conclusions about their impact. Therefore, to help understand the details of a dependency, the aliased objects depended on in the feature need to be made explicit and put into the context in which they are used.

The challenge we face is that there are potentially hundreds of objects on which a feature depends. Thus, it is difficult to understand the dependencies in isolation. What is missing are the relationships between objects. If we view object dependencies in a larger context we can interpret them more easily and attribute semantic meaning to them.

We observed that most objects a feature depends on reference each other. This is plausible because, to use an object, a feature often needs to access another object that holds a reference to it. Consequently, the feature also depends on the object from which the reference was accessed. The exceptions are objects referred to from outside the application, for example from the GUI framework or the program's main method.

Based on these reference relationships of the object dependencies, we build the *object dependency graph*. Figure 5.4 illustrates such a graph taken from the *Receive Message* feature of the IRC chat client case study. Each node represents an object depended on in the feature. Each edge represents an object reference that is subject to a dependency — that is, the reference is accessed in the selected feature but created in another one. In other words, the object dependencies of a feature directly map to the references (edges) shown in the graph.

We use grayscale to convey information about which feature an object or a reference was created in. Light gray means an object or reference was created in a feature that was exercised early in the program run — dark gray in a more recent one. We apply a force-based layout algorithm to visualize the graph.

In the prototype implementation of this visualization, the class name of an object is shown in a tooltip when moving the mouse over it. Accordingly, the tooltip of a reference shows in which feature it was created. Furthermore, the user can navigate to the source code in which an object is used to obtain additional information about the context in which dependencies occur.

In the following section, we present the object dependency graph in more detail on two case studies.

5.5 Case Studies

To evaluate our approach, we structure the discussion of the case studies based on the following two questions:

1. *How many indirect dependencies exist compared to direct dependencies (*cf. Salah's approach*) and how relevant are the indirect dependencies for software maintenance?* This first question is supposed to answer whether indirect dependencies introduced by object aliasing — which our approach additionally detects — actually exist and if so how relevant they are. To answer this question we implemented the approach of Salah *et al.* [SALA 04] to compare the resulting dependencies with ours.

2. *How was the part of the object graph on which a feature depends stepwise modified by the previously exercised features?* In this second question we investigate how the proposed visualization supports a software engineer to explore dependencies to detect possibly hidden connections between the code of the feature he is modifying and other parts of the system.

5.5.1 IRC Chat Client

As a first case study we chose an IRC Chat Client. Our motivation was (1) because it is a small (39 classes and 1063 methods) but non-trivial legacy application and (2) we have access to the source code. A total of six developers contributed to the project which underwent various refactorings and enhancements over nine years.

How many indirect dependencies exist compared to direct dependencies and how relevant are the indirect dependencies for software maintenance? We exercised nine distinct features. Figure 5.5 shows the features in the order they were run from top to bottom with the number of dependencies they have. The feature *Startup* naturally does not have dependencies because it is run first. In all subsequent features our analysis found dependencies upon previous features.

Figure 5.5 also provides the number of dependencies of Salah's approach. It shows that compared to their approach, we detect more dependencies in the *Connect* feature and all subsequent features. In the second feature, *Setup*, we found exactly the same number of dependencies. This is what we expected as this feature was the second feature we exercised, so it cannot depend on more than one feature.

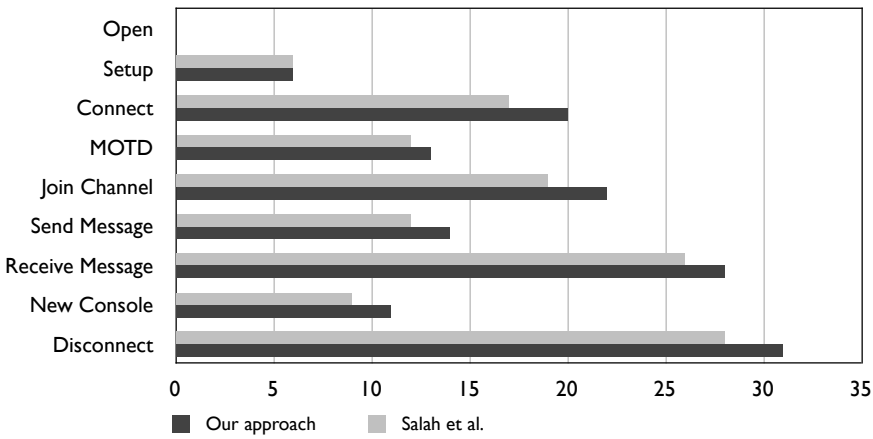

Figure 5.5: IRC features and number of dependencies.

We now present some anecdotal evidence indicating that the additionally found dependencies play a central role among the other dependencies of a feature for maintenance.

For instance, the *Connect* feature depends on both the *Open* and the *Setup* feature with respect to the connection instance. The connection was created in *Open*, hence, Salah's approach only finds this dependency. This rises the question why the feature *Connect* depends on the feature *Setup* with respect to the connection instance?

A closer investigation showed that the objects created for the setup dialog of the *Setup* feature were still alive and used in the feature *Connect* (the activity we can observe with these objects is that they regularly send messages to the connection instance to check its connection state). The

conclusion we can draw from this is that closing the setup window did not clean up correctly and hence left instances behind which continued to be used although they were not needed anymore. The same dependency also occurs in all subsequent features and hence contributes to the number of additional dependencies.

Another additional dependency is the one discussed as an example in Section 5.3 (see Figure 5.3). It illustrates that *Receive Message* depends not only on *Startup* but also on *Join Channel* because the latter feature creates an alias to the connection object. Therefore, when modifying behavior related to the connection in the feature *Receive Message*, the developer not only needs to carefully look at the implementation of the feature *Startup* where the connection object is created, but also at the implementation of *Join Channel*, where the connection is aliased.

A similar dependency like the one described above exists between *Send Message* and *Join Channel*. Both cases reflect a domain constraint: sending and receiving messages takes place in an IRC channel. Therefore, the implementation of sending and reading a message depends on the implementation of joining a channel. This dependency is not directly obvious from the source code because the channel object is not created when joining a channel but at startup.

How was the part of the object graph on which a feature depends stepwise modified by the previously exercised features? To support the interpretation of runtime dependencies of a feature, we evaluated the usefulness of the object dependency graph visualization. As a concrete example, we discuss the dependencies of the *Receive Message* feature. We already illustrated its object dependency graph in Section 5.4. Figure 5.6 presents a part of this graph with annotations of instances and messages sent to them in the feature (in our tool we can access this information interactively on demand).

Let us consider the long loop starting at the IRCConnection object. The connection holds a dictionary that maps channel names to channels (instances of IRCChannelInfo). The grayscale of the channel object indicates that the channel was not created in the same feature like the dictionary in which it is contained (in *Connect*), but is created later in the *Join Channel* feature.

The object dependency graph reveals the composition hierarchy of objects depended on in the context of the feature. Inspecting the message sends further helps us to map the object dependencies to the runtime behavior of the feature.

Based on our analysis, we extract and reconstruct the following activity in the *Receive Message* feature. First, the connection gets the appropriate channel for the message received (see messages subscribedChannels and at:). Then the channel iterates over the set of subscribers to notify them

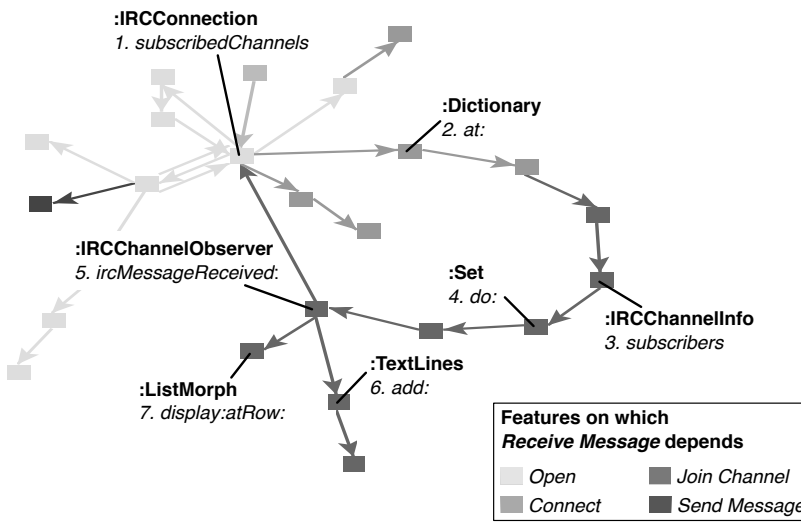

Figure 5.6: Object dependency graph of *Receive Message* feature annotated with invoked methods.

of a received message (ircMessageReceived:). The observer adds the new message to its text lines object which is the model of the UI list widget being updated with display:atRow:.

Summarizing, the object dependency graph reveals the following key information about the runtime dependencies of a feature:

- How many and which object dependencies originate in a feature, and whether this feature was exercised recently or earlier in the program execution. Figure 5.6 shows that most objects are created in three stages: in the *Open*, *Connect*, and *Join Channel* features (only one object dependency exists on the *Send Message* feature).
- For each object dependency, the incoming references show through which other object(s) the object was accessed while the feature was exercised. The brightness of a reference indicates in which feature the reference was created. In Figure 5.6, for example the list morph and text lines objects are only accessed through the channel observer, whereas the connection received messages from multiple objects.
- It shows object dependencies that served as starting points to further extensions of the object graph in a later run feature. An example is the connection which stores a dictionary of channels. This dictionary is not created in the same feature as the connection but later in the *Connect* feature.

- It shows object aliasing, *i.e.*, an object referred to by more than one other object. The software engineer can spot aliases of an object that are created in a later feature than the one that created the object. For example the connection instance is aliased. We can see that some aliases were created in the same feature (*Open*) as the connection itself. Two aliases to the connection, on the other hand, were created in later features.

The aliasing situation described in the last point is particularly interesting because it allows one to visually spot dependencies on objects referenced in one feature but created in another. These are exactly the cases which our approach is capable of detecting because it does not only consider object creation and usage but also tracks aliases.

Summary of results. The case study showed that Salah *et al.* indeed captures most of the feature dependencies. However, the additional dependencies that we uncover are precisely the indirect feature dependencies that can be problematic during maintenance. In one case, a dependency even pointed out an anomaly of the program.

The object dependency graph visualization proved to be of great help for our analysis — it was much simpler than if we had looked at the dependencies one by one. The visualization allowed us to get a quick overview of the dependencies of a feature but also helped us to spot interesting dependency situations.

5.5.2 Pier CMS

Pier is a web content management system [RENG 06]. It is a reengineered version of SmallWiki [DUCA 05]. Its core comprises 177 classes and the metamodel 200 classes. Our choice of Pier was motivated by the following reasons: (1) it is open source, (2) we are familiar with the predecessor application SmallWiki, (3) we are familiar with the features of Pier from the user's perspective, and (4) we have direct access to developer knowledge to verify our findings.

How many indirect dependencies exist compared to direct dependencies and how relevant are the indirect dependencies for software maintenance? For our experiment, we traced 11 features as listed in Figure 5.7. The average number of dependencies per feature is much higher compared to the IRC Client case study. Again the first feature does not have any dependencies. The comparison with the approach of Salah *et al.* shows that we found additional dependencies in all features, except for the first two (for the same reason as explained above).

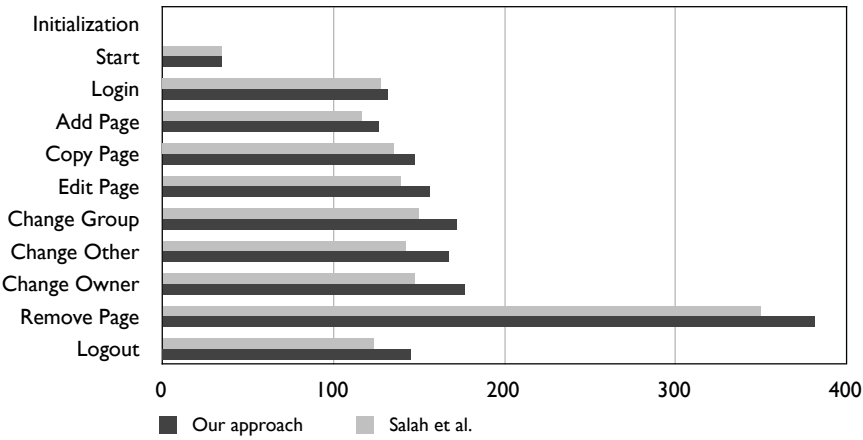

Figure 5.7: Pier features and numbers of dependencies.

All features are related to *Initialization*, the feature that is exercised to load the Pier application, and to the *Start* feature which displays the first web page. This is because these features are responsible for initializing the system, the user session, and its UI components.

As with the IRC Client case study we analyzed the additionally detected dependencies. Again, it turned out that they play important roles in the system.

For example, in each feature exists a dependency on an instance of User. This instance is created during the feature *Initialization*. In the *Login* feature (we logged in as administrator) it is accessed from the kernel and stored in a context object. In each subsequent feature this user object is then used for controlling access. Additionally, in the feature *Change Owner* the user is accessed also from the page on which we are changing the owner. The object dependency graph accurately shows that the reference from the page to the owner was created in the feature *Add Page*. This means, that when the page was instantiated, the user creating it is assigned to be its owner.

How was the part of the object graph on which a feature depends stepwise modified by the previously exercised features? By exploring the object dependency graphs we notice that some of the dependencies are recurring (*i.e.*, the same dependencies exist in most of the features). The dependencies are due to the nature of the Pier application as a web application. Thus every feature makes use of page rendering activity. For example, the *Login* feature reveals similar dependencies to the *Start Page*. This is due to the page being redisplayed after the login action has completed. This char-

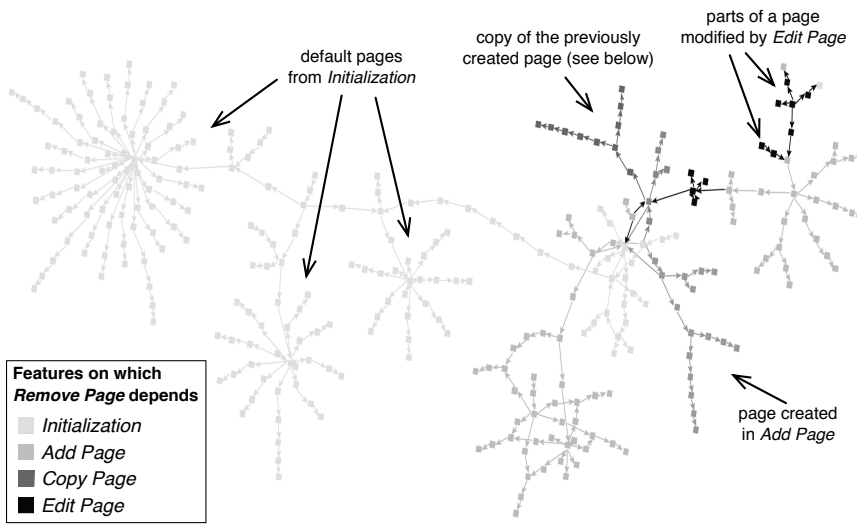

Figure 5.8: Object dependency graph of the *Remove Page* feature from Pier.

acteristic of Pier explains the much higher average number of dependencies compared to the IRC Client case study.

A revealing observation was that the object dependency graph of most features reflects the hierarchical structure of pages. Pier has a very fine-grained object model to represent content (*e.g.*, lists are composed from list items, each containing text or link objects). Behavior like page rendering or copying is performed by Visitors which traverse the full object trees. Hence, there exist a large number of dependencies on the features that create or copy pages.

The order of exercising the features also impacts the dependencies. For example, each feature accesses a PRContext instance created by the previously exercised feature.

The *Remove Page* feature stands out in Figure 5.7 with a much larger number of dependencies compared to the other features. This surprised us because we expected that removing a page would be a rather small feature. Figure 5.8 illustrates the object dependency graph of the *Remove Page* feature. It contains large trees representing all existing pages in the system. There are default pages at the left side and smaller pages at the right. The latter are created by *Add Page* and *Copy Page*. A closer investigation showed that when the Pier application removes a page, it iterates over the entire structure to check for the existence of links to the page that is to be removed. This activity generates a lot of dependencies.

Summary of results. The Pier application showed very different characteristics compared to the IRC Client. It comprises a much larger number of dependencies. There are two reasons. (1) being a web application the contents of a web page is re-generated on each request, and (2) the domain model of Pier is larger in terms of the number of objects created. In both case studies our approach detected additional dependencies in all features except for the first two, and the relative number of additional dependencies is similar.

5.6 Related Work in Feature Analysis

Our work is directly related to the fields of feature related research [BALL 99, EISE 03, MEHT 02, GREE 05].

Our work builds on the analysis of runtime feature relationships, pioneered by Salah and Mancoridis [SALA 04]. Their approach built on well-established dynamic analysis *Feature Identification* techniques (*e.g.*, *Software Reconnaissance*) [WILD 95] to extract features. They defined a hierarchy of dynamic views which track inter-feature dependencies. As discussed in Section 5.2.2, their main definition *depends* detects situations where an object is used in a different feature than the one it is created in. We have shown why this definition misses dependencies and we propose a more precise notion of feature runtime dependencies.

Kothari *et al.* [KOTH 06] proposed an approach to system comprehension that considers features as the primary unit of analysis. They define a relationship between features based on comparing the implementations of two features in terms of the executed methods. They model a feature as a call graph and use graph algorithms to determine the similarities between pairs of features. Also other approaches are based on an analysis of the executed methods, *e.g.*, for locating features in the source code [EISE 05b].

These approaches analyze the dynamic behavior of a system at the granularity of methods. For our problem of detecting runtime feature dependencies, however, tracing method executions alone is not enough. A feature potentially has a dependency relationship on another feature even without executing the same methods. Our approach detects the dependency that occurs when a feature stores an object in a field and a feature exercised later reads this field in a different method.

Various approaches have extended method tracing to improve object-oriented program understanding. For example, apart from Salah *et al.* [SALA 04] mentioned above, Antoniol *et al.* [ANTO 05] consider instance creation events to locate features. In contrast, our Object Flow Analysis is much more radical as it proposes a new model that is centered around objects and tracks the transfer of their references.

Related to the analysis of object references are query-based debugging approaches, which let the programmer test relationships between objects [GOLD 05, LENC 99, DUCA 06]. In contrast to our approach, the query-based approaches are more suited to finding inconsistencies in object graphs than detecting feature dependencies. Also, *a priori* knowledge about the implementation is required to be able to write queries whereas our approach can be used to study an unfamiliar system.

Related work on Shape Analysis helps developers to investigate object graphs to analyze the shape of object structures in memory snapshots (for details see Chapter 2). These approaches face the difficulty that object graphs can be very large. In contrast, this is less of an issue for our approach because the set of objects a feature depends on is typically a relatively small subset of the complete object reference graph. Our approach provides a focused view by only showing those objects and references relevant for the selected feature under investigation. A novel concept of our approach is the notion of time encoded in a grayscale scheme, which reveals in which feature an object or a reference was created.

5.7 Summary of the Chapter

In this chapter we analyze the problem of runtime dependencies between features in an object-oriented system. Our approach builds on previous work by Salah *et al.* [SALA 04]. The essential difference between Salah's approach and our approach is that the former detects dependencies by identifying objects and the features they are created in, whereas our approach identifies object references and how references are transferred between features. Taking object aliasing into account, we propose a novel detection strategy for feature runtime dependencies that additionally exposes indirect dependencies.

Based on the Object Flow Analysis metamodel and our framework to reason about dependencies, the analysis of runtime feature dependencies can be concisely expressed. That is, the only definitions we have to add are the relation that maps aliases to features and the filtering of the relevant aliases of a feature. With these two definitions, the predefined notion of indirect dependencies exactly matches the desired notion of dependencies for features.

Chapter 6

Test Blueprints

Exposing Dependencies in Execution Traces

Writing unit tests for legacy systems is a key maintenance task. When writing tests for object-oriented programs, objects need to be set up and the expected effects of executing the unit under test need to be verified. If developers lack internal knowledge of a system, the task of writing tests is non-trivial. To address this problem, we propose an approach that exposes side effects detected in example runs of the system, and that uses these side effects to guide the developer when writing tests. We introduce a visualization called *Test Blueprint* through which we identify what the required fixture is and which assertions are needed to verify the correct behavior of a unit under test. To demonstrate the usefulness of our approach we present results from two case studies.

6.1 Introduction

Creating automated tests for legacy systems is a key maintenance task [DEME 02]. Tests are used to assess if legacy behavior has been preserved after performing modifications or extensions to the code. Unit testing (*i.e.*, tests based on the XUnit frameworks [BECK 98]) is an established and widely used testing technique. It is now generally recognized as an essential phase in the software development life cycle to ensure software quality, as it can lead to early detection of defects, even if they are subtle and well hidden [BERT 07].

The task of writing a unit test involves (i) choosing an appropriate program unit, (ii) creating a *fixture*, (iii) executing the unit under test within the context of the fixture, and (iv) verifying the expected behavior of the unit

using *assertions* [BECK 98]. All these actions require detailed knowledge of the system. Therefore, the task of writing unit tests may prove difficult as developers are often faced with unfamiliar legacy systems.

Implementing a fixture and all the relevant assertions required can be challenging if the code is the only source of information. Object aliasing and complex chains of message sends hide how and where side effects are produced [BERT 07]. Developers usually resort to using debuggers to obtain detailed information about the side effects, but this implies low-level manual analysis that is tedious and time consuming [ZELL 05].

Thus, the underlying research question of the work we present in this chapter is: *how can we support developers faced with the task of writing unit tests for unfamiliar legacy code?* The approach we propose is based on analyzing runtime executions of a program. Parts of a program execution, selected by the developer, serve as examples for new unit tests. Instead of manually stepping through the execution with a debugger, we use Object Flow Analysis to derive information to support the task of writing tests without requiring a detailed understanding of the source code.

In our experimental tool, we present a visual representation of the dynamic information in a diagram similar to the UML object diagram [FOWL 03]. We call this diagram a *Test Blueprint* as it serves as a plan for implementing a test. It reveals the minimal required fixture and the side effects that are produced during the execution of a particular program unit. Thus, the Test Blueprint reveals the exact information that should be verified with a corresponding test.

To generate a Test Blueprint, we need to accurately analyze object usage, object reference transfer, and the side effects that are produced as a result of a program execution. To do so, we apply Object Flow Analysis in conjunction with conventional method execution tracing. We have implemented a prototype tool to support our approach and applied it to two case studies to assess its usefulness.

The main contributions of this chapter are to show (i) a way to visually expose side effects in execution traces, (ii) how to use this visualization, the Test Blueprint, as a plan to create new unit tests, and (iii) a detection strategy based on our Object Flow metamodel.

Structure of the chapter. This chapter is organized as follows. Section 6.2 discusses difficulties of writing unit tests in the reengineering context. Section 6.3 explains the analysis of object flow in execution traces and Section 6.4 introduces the Test Blueprint, which is based on this analysis. Section 6.5 presents our approach in detail and Section 6.6 discusses two case studies. In Section 6.7 we discuss related work in testing and we conclude the chapter with Section 6.8.

6.2 The Challenge of Testing Legacy Code

To illustrate the task of writing unit tests for unfamiliar code, we take as an example system the Smalltalk bytecode compiler introduced in the previous chapters.

The following code (written in Smalltalk) illustrates a test we would like to generate for the addTemp: method of the FunctionScope class of our Compiler example. During the AST to IR transformation phase of compilation, variables are captured in a scope (method, block closure, instance, or global scope). The temporary variables are captured by the class FunctionScope, which represents method scopes.

```
function := FunctionScope new.
name := 'x'.

var := function addTemp: name.

self assert: var class = TempVar.
self assert: var name = name.
self assert: var scope = function.
self assert: (function tempVars includes: var).
```

Without prior knowledge of a system, a test writer needs to accomplish the following steps to write a new test:

1. **Selecting program unit to test.** When writing tests for a legacy system, the developer needs to locate appropriate units of functionality — that is, a unit which is currently not already covered by a test and is not too large a unit for which to write a test.

2. **Creating a fixture.** To create a fixture, the developer has to find out which objects need to be set up as a prerequisite to execute the behavior under test. In this example, we need to create a FunctionScope instance, which is used as the receiver, and a string, which is used as the argument of the message send addTemp:. Creating this fixture is straightforward. If more objects need to be set up, however, it may be difficult to understand how they are expected to reference each other and how to bring them into the desired state. Incorrectly set up objects may break or inadvertently alter the behavior of the unit under test.

3. **Executing the unit under test.** Once we have the fixture, this step just involves executing the method, in our example addTemp:, using the appropriate receiver and arguments from the fixture. The execution of the program unit stimulates the fixture, that is, the execution returns a value and produces side effects.

4. **Verifying expected behavior.** We need to know what the expected return value and the side effects are. In our example, the returned object is expected to be a new TempVar instance with the same name 'x'. Furthermore, the returned TempVar should reference the FunctionScope object that we used as receiver. And finally, the FunctionScope should include the returned object in its tempVars collection. It is difficult to detect which side effects have been produced as a result of a program execution, as this information may be obscured in complex chains of method executions. By browsing the source code, it is difficult to ascertain this information. And although a debugging session reveals the required information, this may be a tedious approach in a large and complex system.

6.3 Applying Object Flow Analysis

The Test Blueprint visualization presented in this chapter provides information about a small part of the execution trace, which we refer to as an *execution unit*. An execution unit is a set of method invocations selected by the developer and represents a unit of behavior for which the developer wants to write a unit test.

The analysis applied in this chapter exploits the dependencies introduced by the object reference transfer between the selected execution unit and the rest of the execution trace. To reason about these dependencies, the set of $Regions$ therefore contains only two elements. The first element is $Unit$, a set of method invocations selected by the developer, and the second element is $Unit^C$, the set of all method invocations *not* contained in the unit (see Table 6.1).

$a \in Aliases$	*(set of aliases created at runtime)*
$i \in Invocations$	*(set of method invocations)*
$i \in Unit \subset Invocations$	*(set of selected method invocations)*
$i \in Unit^C = Invocations \backslash Unit$	*(complement — all other method invocations)*
$r \in Regions = \{Unit, Unit^C\}$	*(set of regions)*
$reg : Aliases \rightarrow Regions$	*(region in which an alias resides)*
$aliases : Regions \rightarrow \mathcal{P}(Aliases)$	*(set of relevant aliases of a region)*

Table 6.1: Sets and relations used for the Test Blueprint analysis.

We impose the following condition on the structure of units, which states that units are sub-traces of execution traces. Let $allCallees$ be defined as the transitive closure of the $callees$ relationship of a method invocation i,

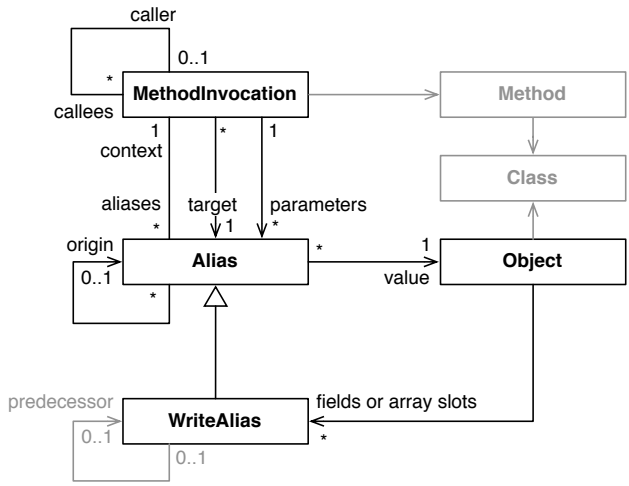

Figure 6.1: Object Flow Analysis metamodel (entities and associations exercised by the Test Blueprint analysis are highlighted in black).

extended to sets of invocations. Then $Unit$ satisfies:

$$\exists i \in Unit \quad \text{such that} \quad \{i\} \cup allCallees(i) = Unit$$

Intuitively, the condition verifies that the set $Unit$ consists of a method invocation and all invocations that this method transitively calls. In other words, in a $Unit$ there exists exactly one method invocation, which we refer to as *root method invocation*, that has a caller that is not in the $Unit$.

To complete the definitions required by our framework we define the relation reg as follows.

Definition 10 *(Region of alias)*

$$reg(a) := r \in Regions \quad \textit{such that} \quad a.context \in r$$

This definition reveals whether an alias is created inside or outside of the execution unit. The region of an alias is $Unit$ if the alias is created in a method invocation contained in $Unit$, else the region is the set $Unit^C$. (Its worth to note that the containment relationship is unambiguous because $Regions$ is a partition of the set of method invocations).

Analyzing the fixture. To show how the fixture for a unit under test needs to be set up, the Test Blueprint indicates the object state that the execution

unit depends on — that is, the objects and their interrelationships that are expected to exist such that the execution of the method under test exhibits the same behavior as in the analyzed example run of the program. We can express this expected state by the dependency of the region $Unit$ on its complementary region (notice, that there is no difference between direct and indirect dependencies since there exist only two regions).

Definition 11 *(Fixture)*

$$\mathit{fixture} := \{a \in \mathit{aliases}(\mathit{Unit}) : \mathit{reg}(a.\mathit{origin}) \neq \mathit{Unit}\}$$

This definition is a reformulation of DD, which relates two regions. For the Test Blueprint, however, we need more detailed information than the one provided by DD. Therefore we define $\mathit{fixture}$ to yield a set of aliases on which the behavior of the execution unit depends.

The set $\mathit{fixture}$ contains three kinds of aliases that represent the transfer of an object into the unit: (i) by reading a field or array slot, (ii) by passing a reference as argument of the root method, or (iii) by the target reference of the root method (referred to by this in the method).

Notice that it is not possible to pass a reference into a unit by a method return value because of the sub-trace condition of units stated above. Also, write aliases do not occur because they are always created in the same method as the alias from which they originate.

Analyzing effects. Analogous to the analysis of the fixture, the Test Blueprint exposes the expected (side) effects of executing the unit under test. The logical definition of the effects produced is to detect how $Unit^C$ depends on $Unit$:

$$\mathit{effects} := \{a \in \mathit{aliases}(\mathit{Unit}^C) : \mathit{reg}(a.\mathit{origin}) \neq \mathit{Unit}^C\}$$

This definition yields the aliases that are created in the execution unit and that are later used outside of it. However, this definition is too conservative as it only detects side effects that actually influence the later program execution. This means, a side effect is not detected if a field or array write alias is not read again later on.

The following definition is less restrictive as it yields all side effects (write aliases) created in the execution unit, regardless of whether the reference is used afterwards in the analyzed program execution.

Definition 12 *(Effects)*

$$\begin{aligned}\mathit{effects} \quad := \quad & \{a \in \mathit{aliases}(\mathit{Unit}) : a\ \mathit{kindOf}\ \mathsf{WriteAlias}\} \cup \\ & \{a \in \mathit{aliases}(\mathit{Unit}^C) : \mathit{reg}(a.\mathit{origin}) \neq \mathit{Unit}^C \wedge a\ \mathit{kindOf}\ \mathsf{ReturnAlias}\}\end{aligned}$$

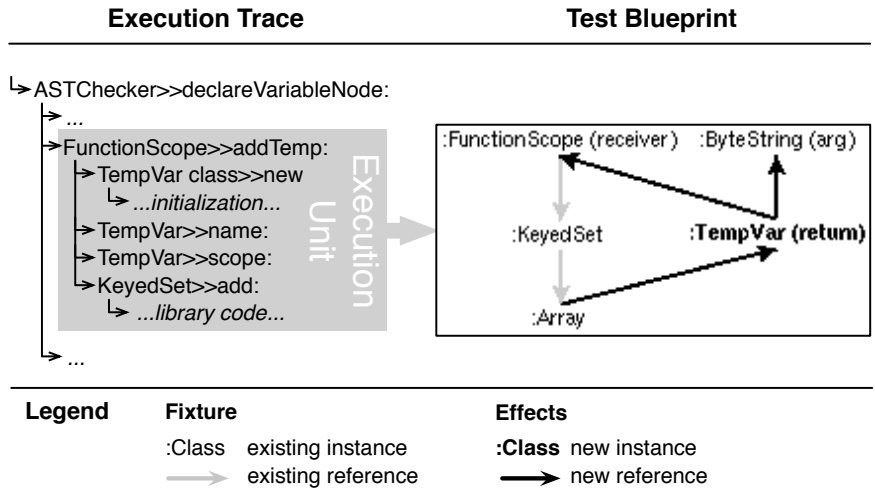

Figure 6.2: An execution unit and the Test Blueprint produced from it.

This definition yields a set of write and return aliases. Write aliases represent the side effects produced in the unit under test, and the return alias (there exists only one) represents the returned value of the root method.

6.4 Introduction of the Test Blueprint

In this section we introduce the Test Blueprint based on the analysis of object reference transfer discussed in the previous section. In Section 6.5 we then present how the Test Blueprint supports writing unit tests by revealing the information required to establish the fixtures and identify the required assertions to verify the correct behavior.

The left side of Figure 6.2 illustrates an excerpt of an execution trace, displaying the method executions as a tree. The sub-tree with the root method invocation FunctionScope»addTemp: represents an execution unit. The Test Blueprint of this execution unit is displayed at the right side of Figure 6.2.

Information provided by the Test Blueprint. The Test Blueprint is similar to a UML object diagram [FOWL 03] in that it shows objects and how they refer to each other. The key difference is that the Test Blueprint is scoped to the behavior of an execution unit and that it also shows (i) which objects were used by the execution unit, (ii) which references between the objects

have been accessed, (iii) what objects have been instantiated, and (iv) what side effects were produced.

This information is encoded as follows in the Test Blueprint. We use regular typeface to indicate objects that existed before the start of the execution unit and bold typeface to indicate objects that are instantiated in the execution unit. The receiver object, the arguments, and the return value are indicated. The visualization shows only objects that have actually been accessed (but not necessarily received messages).

An arrow between two objects indicates that one object holds a field reference to another object (or that an array refers to an object). Like with objects, only references are displayed that have actually been accessed. Gray arrows indicate references that already existed before the execution unit was run.

A gray arrow displayed as a dashed line means that the corresponding reference is deleted during the execution unit. Black arrows indicate references that are established during the execution unit. Thus, the black and dashed arrows represent the side effects produced by the execution unit.

Building the Test Blueprint. The set of aliases that the definition *fixture* yields are used as follow. The binary relation $kindOf$ determines whether an alias is either the direct type or one of the supertypes of a given class (the alias class hierarchy is shown in Chapter 3). We distinguish between three kinds of aliases in this set. Let $a \in$ *fixture*:

- a $kindOf$ ReadAlias, then a denotes a gray edge to an object.
- a $kindOf$ ParameterAlias, then a denotes an object passed as parameter (indicated as arg).
- a is the target alias of the root method invocation, then a denotes the object used as target (indicated as receiver).

Objects in the graph introduced by the above rules are displayed in regular typeface, and edges are displayed in gray.

Analogous, the *effects* definition yields the following two kinds of aliases. Let $a \in fixture$:

- a $kindOf$ WriteAlias, then a denotes a black edge to an object.
- a $kindOf$ ReturnAlias, then a denotes the object passed as return value (indicated as return).

Objects in the graph introduced by the above rules are displayed in bold typeface if they are not referred to by a gray edge (a gray edge indicates that the object already existed in the fixture), and edges are displayed in black.

A gray edge is shown as a dashed line if a black edge exists for the same field of an object or slot of an array (which means that the value of this field was modified and the reference shown in gray does not exist anymore after the execution unit has completed).

Understanding the Test Blueprint. Let us consider again the highlighted execution unit in Figure 6.2, which contains all methods in the sub-tree rooted in FunctionScope»addTemp:. In the execution trace this method is called in the following code.

```
ASTChecker>>declareVariableNode: aVarNode
    | name var |
    name := aVarNode name.
    var := scope rawVar: name.
    var ifNotNil: [ ... ] ifNil: [ var := scope addTemp: name ].
    aVarNode binding: var.
    ^ var
```

As the Test Blueprint in Figure 6.2 shows, the receiver of the addTemp: message is an instance of the class FunctionScope and the single argument is a string. Furthermore, the returned object is a newly created instance of the class TempVar. The Test Blueprint in Figure 6.2 also shows the state of the returned object and what side effect the method execution addTemp: produced. Let us compare it to the implementation of the addTemp: method printed next.

```
FunctionScope>>addTemp: name
1    | temp |
2    temp := TempVar new.
3    temp name: name.
4    temp scope: self.
5    tempVars add: temp.
6    ^ temp
```

In the Test Blueprint we see that a new TempVar instance is created (compare to the code at line 2). The string passed as the argument is stored in a field of the TempVar instance (3). Another side effect is that the new object is assigned a back reference to the receiver (4) and that it is stored in a keyed set of the receiver (5). Eventually, the new instance is returned (6).

In the case of the above example, most information contained in the Test Blueprint could also be obtained manually from the source code without too much effort (although, the successively called methods like name:, scope: and add: need to be studied as well). However, this task would not be so trivial in the case of more complex execution units, which may contain many executed methods and complex state modifications.

Figure 6.3 illustrates four other Test Blueprints from the same domain. The Test Blueprint in Figure 6.3.A does not create new objects but it modifies

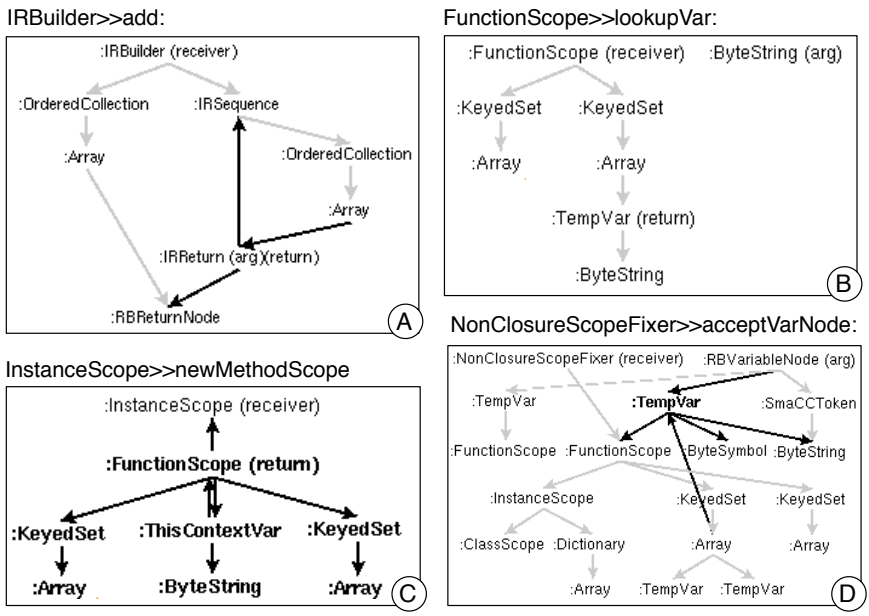

Figure 6.3: Four Test Blueprint examples with different characteristics .

the state of the object passed as the argument and of an existing array, which is accessed indirectly through the object used as receiver. Figure 6.3.B is an example in which no side effects are produced. The execution unit performs a lookup in an existing object structure. In contrast, in Figure 6.3.C only one object is used, and many new objects are created. The Test Blueprint shown in Figure 6.3.D illustrates a case in which an existing object reference is deleted (dashed arrow).

6.5 From the Test Blueprint to Unit Tests

In this section we present our approach and experimental tool to support a developer to overcome the problems stated in Section 6.2 when writing tests. The underlying idea of our approach is to analyze the runtime behavior of a program to find examples for new unit tests. These examples are analyzed and the resulting visualization, we refer to as the Test Blueprint, then serves as a plan for the developer to write a new unit test.

Figure 6.4 provides an overview of our approach. From instrumented program runs we obtain dynamic analysis data about object flow and

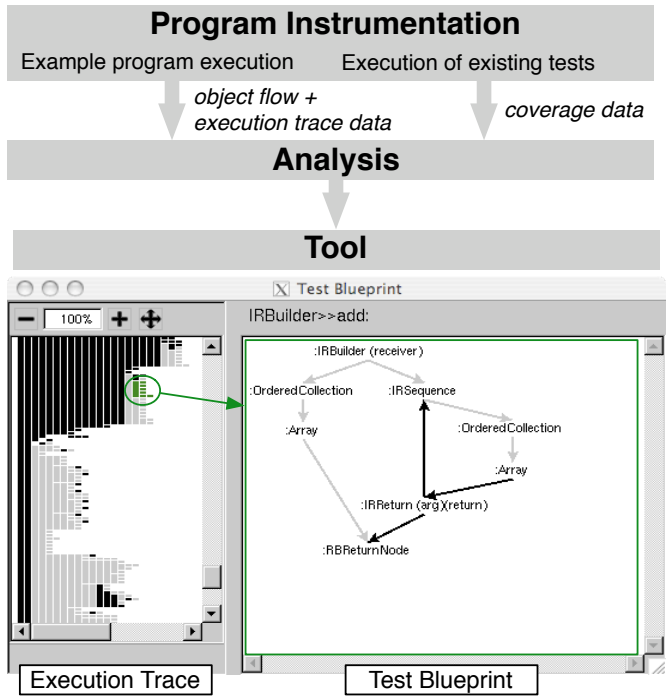

Figure 6.4: Overview of the approach.

method invocations. From an instrumented run of the existing unit tests we record which methods are already covered by tests.

From the developer's perspective, the approach works as follows. In a first step, the developer interacts with our prototype tool to select an appropriate execution unit serving as an example for writing a new test in the execution trace. This is described in detail in Section 6.5.1. The selection causes the Test Blueprint view to be updated. The developer subsequently uses the view as a reference or a plan, providing him with the information necessary to implement the fixture (Section 6.5.2), to execute the unit under test (Section 6.5.3) and to write the assertions (Section 6.5.4).

6.5.1 Selecting a Program Unit to Test

The left view of our tool illustrated in Figure 6.4 shows a filtered execution trace of the program, which was exercised by the developer. The trace is shown as a tree where the nodes (vertical rectangles) represent method

executions. The layout emphasizes the progression of time; messages that were executed later in time appear further to the right on the same line or further down than earlier ones. This view is an adaptation of a view proposed by De Pauw *et al.* [DE P 98], which was later used in the Jinsight tool [DE P 99].

The goal of this view is to provide the developer a visual guide to search for appropriate example execution units in the trace that need to be tested. The tool provides options for filtering the amount of information in the trace based on different criteria, for example to show only the execution of methods of a particular package or a class of the system for which tests should be created.

Additionally, the execution trace is annotated with test coverage information. We compute this information by determining if, for each method in the trace, there exists a test that covers that method. Methods that are not covered are shown in black, whereas methods that are already exercised by a test are shown in gray. With the help of these visual annotations, a developer can more easily locate execution units of untested code on which he needs to focus.

6.5.2 Creating a Fixture

When the developer selects an execution unit, the corresponding Test Blueprint is generated and displayed in the right view of our tool (see Figure 6.4). The selected execution unit (highlighted with an ellipsis) now serves as an example to create a new unit test. To create the fixture of the new test, the Test Blueprint can be interpreted as follows.

First, the Test Blueprint reveals which objects need to be created, namely the ones that are not displayed in bold. Second, the gray references show the object graph — that is, how the objects are expected to refer to each other via field references. (In our tool, the name of the field can be accessed with a tooltip). The created object graph represents a minimal fixture as the Test Blueprint shows only the objects and references that have been accessed in the execution unit.

Unfortunately, it can be difficult to implement the fixture as proposed by the Test Blueprint. The problem is that it is not always obvious how to create and initialize the objects. Often, not only the constructor has to be called with appropriate parameters but also further messages have to be sent to bring the object into the desired state. In some cases, the order in which those methods are executed may also be relevant.

To address this problem, the Test Blueprint provides a means to query for more detailed information about the creation of any of the objects it displays. For each object, we backtrack its flow starting from the location

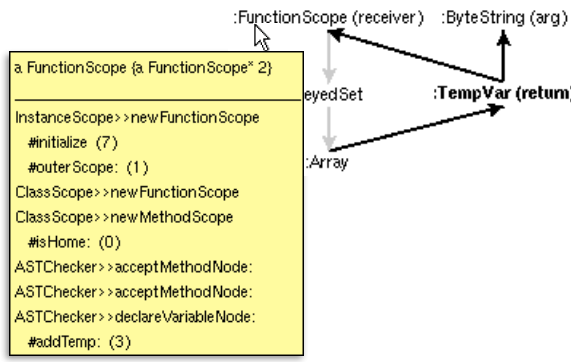

Figure 6.5: Backtracking object setup

where it is first used by the execution unit. Figure 6.5 shows the popup window for the FunctionScope instance.

This view reveals (i) the path of methods through which an object was passed into the execution unit (the top method indicates where the object under investigation was instantiated), and (ii) all messages sent to the object along this path. The number in parentheses indicates how many state modifications were produced, including transitive state.

In Figure 6.5 we see that the FunctionScope object of our running example is instantiated in the method newFunctionScope and that its constructor, initialize, produced seven side effects. In the same method, the execution of outerScope: produces one side effect. No other method execution except for addTemp: modified the object.

Using the backtracking view, we can also find out that the KeyedSet is instantiated in the constructor of FunctionScope. Since no other object state is needed in the fixture apart from the string used as the argument, the following fixture is sufficient.

```
function := FunctionScope new.
name := 'x'.
```

6.5.3 Executing the Unit Under Test

With the fixture created in the previous step, executing the unit under test is straightforward. The Test Blueprint shows which object from the fixture is the receiver and which objects are used as arguments:

```
var := function addTemp: name.
```

6.5.4 Verifying Expected Behavior

In this last step, the test writer needs to verify the expected behavior of the unit under test using assertions. The Test Blueprint reveals which assertions should be implemented:

- The objects shown in bold typeface are the instances that are expected to be created. Thus, assertions should be written to ensure their existence.
- The expected side effects have to be verified: black and dashed arrows between objects denote newly created or deleted field references.
- The Test Blueprint reveals the expected return value.

Once again, we illustrate this with our running example. Here the assertions derived step by step from the Test Blueprint are the following.

```
self assert: var class = TempVar.
self assert: var name = name.
self assert: var scope = function.
self assert: (function tempVars includes: var).
```

The assertions verify that the FunctionScope includes in its tempVars set the returned TempVar instance, and that the back pointer from the TempVar to the FunctionScope exists. Furthermore, the new TempVar is expected to store the string passed as argument.

6.6 Case Studies

In this section we present the results of two preliminary case studies. The first case study provides anecdotal evidence of the applicability of our approach for an industrial system, which supports the daily business of an insurance company. Our main focus with this study is to investigate how well our approach performs in the context of a real world legacy application and to gain experience for future controlled experiments.

In our second case study we applied our approach to a web content management system to evaluate how tests written supported by our approach differ from the tests already present, written by an expert.

6.6.1 Insurance Broker Application

In this case study we wanted to investigate questions regarding the applicability of our approach in a real world scenario, such as: *How hard is it to*

find appropriate execution units in the trace? Can new unit tests be completely implemented following the plan provided by the Test Blueprint? Do tests written with our approach exhibit special characteristics (for example, considering size and complexity)?

Context. The Insurance Broker system is a web based application used both in-house by the insurance company employees as well as remotely by the external insurance brokers. The system has been in production for six years. While the system has been constantly extended over time, its core, which implements the insurance products and their associated calculation models, has not changed much. For the near future, however, a major change affecting core functionality is planned.

One problem associated with this project is that two of the three original developers of this application have left the team and the new members lack detailed knowledge about older parts of the system. At the time we carried out this experiment, the system consisted of 520 classes and 6679 methods. The overall test coverage amounted to 18% (note that we consider only method coverage).

Study setup. To investigate the usefulness of our approach in this context, we had access to a developer to write tests for the application core, which comprises 89 classes and 1146 methods. This developer has only ever worked on newer parts of the system. This meant that he had basic knowledge of the system but was lacking internal knowledge about the core of the system.

As we wanted to ensure that the developer did not have to test any of the code he himself had implemented, we selected a version of the system dating from the time before he joined the team. In the first part of our study we trained him in our experimental tool and demonstrated how to use it to implement a new test. During the following two hours he used our tool to write new tests for the selected part of the system.

Results. The developer quickly understood the principle of the Test Blueprint and how to use it. Within these two hours, the developer created 12 unit tests. With the new tests, the coverage of the core increased from 37% to 50%.

Table 6.2 shows figures from the analysis of the developer's work. The first column labels the tests from 1 to 12. The second column indicates the number of minutes the developer spent to find a new execution unit, to study the Test Blueprint and to implement the test. In the remaining columns we show the following measurements:

- The number of method executions in the selected execution unit.
- The size of the fixture in the Test Blueprint (number of pre-existing objects plus number of gray references). This number is about the same as the number of statements required to set up the fixture.
- The number of side effects in the Test Blueprint (number of new objects plus number of black and dashed references). This number corresponds to the number of assertions.

test #	time[m]	exec. unit	fixture	side eff.
1	13	7	2	5
2	6	13	1	3
3	12	7	4	3
4	6	66	2	2
5	5	1	3	2
6	4	1	2	1
7	4	3	3	5
8	5	35	7	2
9	32	194	21	13
10	5	3	1	2
11	10	201	3	4
12	15	349	10	1
average	10	73	5	4

Table 6.2: Measurements of 12 tests (time in minutes, size of execution unit in number of executed methods, fixture size, number of side effects).

The most complex test the developer created was #9, which tests critical functionality of the system (the calculation of discounts and subsidies). Surprisingly, this functionality was not covered by existing tests. A part of the Test Blueprint of this test is shown in Figure 6.6.

As Table 6.2 shows, this is an exceptional test with respect to the size of the Test Blueprint (the size of the Test Blueprint is the sum of the last two columns). Most tests were created from rather small Test Blueprints. Roughly, the size of the Test Blueprint corresponds to the number of minutes spent implementing the test.

On the other hand, the size of the execution unit (number of executed methods) does not seem to have a direct relationship to the complexity of writing a test. For instance, test #4 has an execution unit of size 66 but only a size of the Test Blueprint of 4. This test exercises the functionality of querying for available products. This involves iterating over all product models and verifying their availability, which caused the 66 method executions.

The largest execution unit is test #12 with 349 method executions. This behavior verifies the validity of a set of products, which involves complex business logic. This test, however, only required one assertion, which is

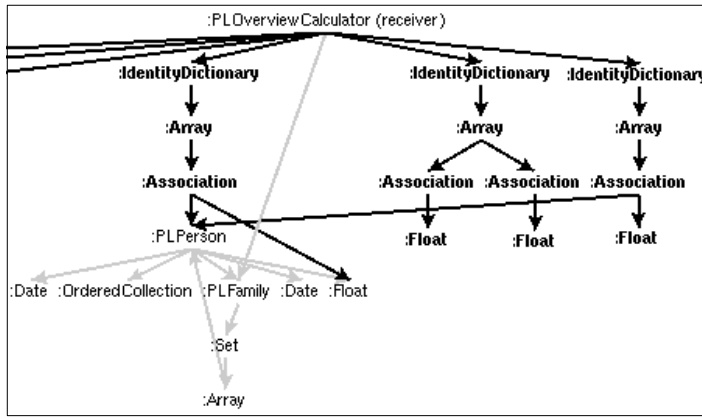

Figure 6.6: Detail of Test Blueprint from test #9

to verify the returned boolean value. Commonly, this test would not be considered a conventional unit test as it exercises much more functionality than what belongs into a unit. Also, this test contributes above average to the increase in the test coverage (which jumped from 37% to 50%).

Observations. One problem we observed was that selecting appropriate execution units in the execution trace is not supported well enough. Although the filtering of relevant methods and the highlighting of uncovered method proved very useful, the developer spent unnecessary time to find execution units that were not too trivial (for example, accessor methods) and not too complex to test.

On the other hand, the Test Blueprint worked very well and as intended. The developer used it as the primary source of information to implement the tests. Yet, sporadically he resorted to consulting the source code, for instance to study how to set up an object. Although the backtracking of object setup helped to indicate what methods to look at, it did not completely replace the activity of consulting the code.

In summary, the developer successfully applied our tool to write tests for a system he only had basic knowledge of. Most of the chosen execution units had rather small Test Blueprints, so that it was generally not a problem to keep track of the objects and references to write the fixture and assertions. Large execution units in the trace did not necessarily indicate large Test Blueprints.

In a future case study, it would be interesting to measure how productive developers using our tool are compared to developers without tool support.

It would also be interesting to see how and which program units are chosen and how the quality of the tests differ. The study we present in the following section, provides initial insights into the differences of the tests.

6.6.2 Web Content Management System

In this case study we wanted to investigate the following question: *How do tests written using our approach differ compared to conventional tests written by an expert of the system?*

For this study, we selected the Pier web content management system [RENG 06]. Its core comprises 377 classes.

Setup. To be able to directly compare tests written using the Test Blueprint with tests written by an expert of the system, we performed the following study. We randomly selected 14 non-trivial unit tests that are shipped with Pier. First we removed all assertions from the source code of the tests (84 in total), leaving only the code for the setup of the fixture and the execution of the unit under test. In the next step we used our approach to analyze the execution of each stripped down test case. Using the guidance of our Test Blueprint, we then systematically rewrote assertions for the tests as demonstrated in Section 6.5.

Results. In summary, the difference of the recreated assertions compared to the original 84 assertions is: (a) 72 of the recreated assertions are identical to the original ones, (b) 12 original assertions had no corresponding assertion in our test, and (c) our tests had 5 additional assertions not existing in the original code.

In 85% of the cases, the assertions we derived from the Test Blueprint were exactly the same as the ones implemented by the main author of the system. But with our approach some assertions were also missed (result b). A closer investigation revealed that most of the missing 12 assertions verify that special objects are left unmodified by the unit under test. The focus of our approach is the side effects, that is, the modified state. Writing assertions to verify that special program state is left unchanged requires in-depth knowledge of the implementation. Our approach does not provide hints which unmodified objects would be worthwhile to verify.

The last result (c), shows that we found additional assertions not existing in the original tests. For instance, one of those assertions tests whether the state of the object passed as argument is correctly modified. The developer of Pier confirmed that, indeed, these relevant assertions were missed (and that he plans to integrate them).

6.7 Related Work in Testing

Specification based testing. There is a large body of research on automatically generating tests or test input from specifications and models [UTTI 06]. In the work of Boyapati *et al.*, they present the Korat framework which generates Java test cases using method preconditions [BOYA 02]. Compared to our approach, these test generation tools require *a priori* specifications, which often do not exist for legacy systems. For our approach, the code and the running system are the only required sources.

Automatic testing. Fully automated testing tools exist such as DART [GODE 05]. DART performs random testing and employs dynamic analysis to optimize coverage. In contrast, our approach analyses real program execution that has been initiated by the developer to create example scenarios for which to write tests. The result of applying our approach are conventional unit tests. In contrast to the automated testing approaches, the intent of our approach is not to achieve a full branch coverage as required by McCabe's structured testing criterion [WATS 96].

Trace based testing. Testlog is a system to write tests using logic queries on execution traces [DUCA 06]. Testlog tackles the same problem as our approach, however, it does so in a different way. The problem of creating a fixture is eliminated by expressing tests directly on a trace. With our approach, the developer creates conventional unit tests without the need to permanently integrate a tracing infrastructure into the system to be able to run the tests.

Other query based approaches, primarily targeted for debugging, can also be used for testing [LENC 99, GOLD 05]. In contrast to our approach, the query based approaches are tailored towards finding inconsistencies rather than to support writing new tests because these techniques require *a priori* knowledge about the source code to write queries.

6.8 Summary of the Chapter

In this chapter we present an approach to support developers in writing unit tests for an unknown system. We propose the Test Blueprint, which provides a plan to write a fixture and assertions for an observed example execution of a unit.

The Test Blueprint requires an analysis of the program state on which the execution of a unit depends (for the fixture), and of how this execution then affects the program state (for assertions). This analysis can be concisely

expressed with Object Flow Analysis. Since the analysis requires details about the references transferred, we reformulate the direct dependency definition provided by our framework to yield the set of aliases for two dependent regions (rather than just relating two regions).

Chapter 7

Practical Back-in-Time Debugging

Effective Object Flow Analysis in the Virtual Machine

Back-in-time debuggers are extremely useful tools for identifying the causes of bugs. Unfortunately the "omniscient" approaches that try to remember *all* previous states are impractical because they consume too much space or they are far too slow. Several approaches rely on heuristics to limit these penalties, but they ultimately end up throwing out too much relevant information. In this chapter we propose a practical approach that attempts to keep track of only the relevant data. In contrast to other approaches, we keep object history information together with the regular objects in the application memory. Although seemingly counter-intuitive, this approach has the effect that data not reachable from current application objects (and hence, no longer relevant) is garbage collected. Following the concept of Object Flow Analysis, we extend the memory model of virtual machines to seamlessly integrate historical execution data. We describe the technical details of our approach, and we present benchmarks that demonstrate that memory consumption stays within practical bounds. Furthermore, the performance penalty is significantly less than with other approaches.

7.1 Introduction

When debugging object-oriented systems, the hardest task is to find the actual root cause of a failure as it can be far from where the bug actually manifests itself [ZELL 05]. In a recent study, Liblit *et al.* examined bug symptoms and found that in 50% of the cases the execution stack contains essentially no information about the bug's cause [LIBL 05].

Classical debuggers are not always up to the task, since they only provide access to information that is still in the runtime stack. In particular, the information needed to track down these difficult bugs includes (1) how an object reference got here, and (2) the previous values of an object's fields. For this reason it is helpful to have previous object states and object reference flow information at hand during debugging. Techniques and tools like back-in-time debuggers, which allow one to inspect previous program states and step backwards in the control flow, have gained increasing attention recently [LEWI 03, POTH 07, HOFE 06, MARU 03, POTH 08].

The ideal support for a back-in-time debugger is provided by an *omniscient* implementation that remembers the complete object history, but such solutions are impractical because they generate enormous amounts of information. Storing the data to disk instead of keeping it in memory can alleviate the problem, but it has the drawback of even further increasing the runtime overhead. Current implementations such as ODB [LEWI 03], TOD [POTH 07] or Unstuck [HOFE 06] can incur a slowdown of factor 100 or more for non-trivial programs.

The common strategy for discarding data is to delete the oldest data first, which inevitably leads to the problem that bugs that have their cause located far enough from their effect cannot be tracked down anymore [LEWI 03]. Another strategy to address the memory problem is to generate less data by only instrumenting parts of the application [POTH 07]. In this case, however, the programmer must know upfront where the potential source of the problem is. This approach produces less data, but it still presents the problem that the data grows over time making it necessary to discard old data at some point. Unfortunately, the common technique of logging a sequential trace of events does not offer much better possibilities than to delete the oldest data first.

Object Flow Analysis offers a solution by capturing more details about the execution history than execution traces typically do. By representing the connections between references as alias relationships, a more expressive notion of relevance can be defined. In our model, the relevance of an alias is given by whether the alias is reachable from the root objects at a given point in time. For example, when a method invocation has returned and the parameter passed to it was not assigned to a field, both the method invocation and the parameter alias can be discarded.

Initially, we started investigating this idea by using the origin and context relationships as defined in the Object Flow Analysis metamodel. The first results were not very promising because too much data was thrown away. Although all previous steps in the flow of an object, including the call stacks, were retained, in some cases it was not possible to go back in the history of a field because the object previously stored in the field had already been discarded.

Indeed, the notion of object *reference structure* was missing. The solution to capture also reference structure in our metamodel turned out to be surprisingly simple to realize: by linking the current write alias of a field to the previous write alias of the same field, the state history of the field is retained. In the Object Flow Analysis metamodel, this link is the predecessor association between write aliases. Now, even though an object may no longer be referenced, it is not discarded as long as another object exists that once had a field reference to the object in question.

Having solved this problem, the remaining challenge is to make the detection of irrelevant history very fast, because it has to be performed while the program is running. The back-in-time debugging approach we present in this chapter proposes to implement the Object Flow Analysis metamodel at the level of the virtual machine — that is, to keep the recorded aliases in the same memory space as the regular application objects. In this way we seamlessly integrate historical execution data into the object model of the virtual machine, rather than tracing isolated events and store them in a log. A direct consequence of this approach is that aliases no longer reachable from the objects of the running application will be automatically garbage collected.

This design provides the following benefits:

- The relevance of an alias and method invocation is determined by the reachability in the memory graph. Which history and how much of it is retained depends on the interconnectivity of the aliases and method invocations.
- Garbage collection of the recorded history comes "for free" since we can employ the usual garbage collector without any modifications to incrementally and efficiently delete no longer reachable data.

As our evaluation shows, how much memory is consumed with our approach largely depends on the characteristics of the application. In some cases the data recorded does not grow indefinitely and hence in these cases recording can be turned on all the time. However, our approach does *not* guarantee that the virtual machine will never run out of memory — it only makes it less likely. In case the recorded data continues to accumulate over time, we run out of memory much later than with conventional approaches.

In the latter case, we provide means to configure the recording to capture and remember less data, which can lead to a dramatic decrease in memory consumption.

To make back-in-time debugging truly practical, it is important not only to manage memory consumption, but also to keep the runtime overhead within reasonable limits. A slowdown of 100 can make a program unusable even for debugging. Unlike many other back-in-time debuggers, which rely on bytecode manipulation techniques and application-level logging, our implementation is at the virtual machine level and because of that the performance is significantly improved. From our experiments the worst case scenario led to a slowdown of only a factor of 7 compared to the original virtual machine.

Chapter structure. In the following section we describe our approach to incorporate the Object Flow Analysis metamodel into a high-level language virtual machine. In Section 7.3 we discuss our implementation, in Section 7.4 we present our evaluation and in Section 7.5 we discuss the results. We present the related work in Section 7.6 and we conclude by summarizing the chapter in Section 7.7.

7.2 Approach: An Object-Flow-Aware VM

Most back-in-time debuggers are based on tracing events emitted at the application level. This technique is commonly based on transforming bytecode to introduce sensors that emit events. We take a radically different approach by modifying the virtual machine to add the program's execution history to the object model — following the metamodel of Object Flow Analysis.

7.2.1 Representing References in Memory

The memory layout of objects in object-oriented virtual machines typically consists of a header for the class pointer, hash bits, GC flags, size, etc. and a fixed number of fields containing object references and primitive values. In many virtual machines, object references are implemented as direct pointers — that is, an object reference is just the address of that object in memory. Examples are the Sun HotSpot VM, Jikes RVM (formerly known as the Jalapeño VM [ALPE 99]), and the Squeak Smalltalk VM [INGA 97].

In the proposed object model we add a level of indirection by representing object references by *alias* objects.

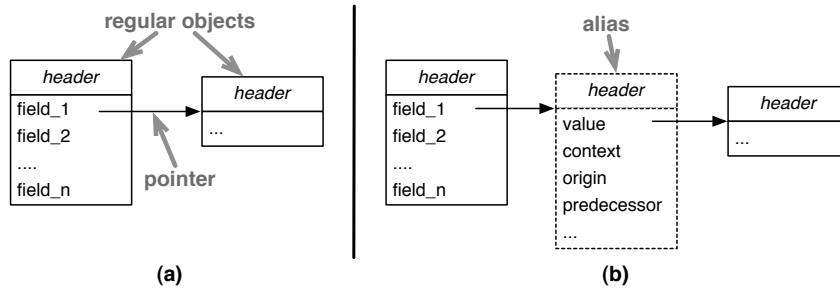

Figure 7.1: (a) Typical object format with references as direct pointers and (b) proposed extension with references being optionally represented by alias objects.

Figure 7.1.a illustrates the typical approach where an object reference is represented by a pointer, and Figure 7.1.b shows how in our object memory model the object reference is substituted by an alias object. Thus, the pointer stored in field_1 points to the alias and the alias has a pointer to the actual object. Aliases cannot be nested; the object reference of an alias is always a direct pointer to a non-alias object. Aliases cannot only substitute reference values but also the undefined value and primitive values. In this case field_1 contains the primitive value (*e.g.*, tagged pointer for small integers).

Aliases have the following key properties that distinguish them from common objects:

- **Transparency.** Aliases are completely invisible at the application level. This means that the semantics of the language are not altered. For instance, method lookup, field access, or primitive operations are performed as if the actual object were referenced directly. To make the information of an alias accessible at the application level we use the concept of Mirrors [BRAC 04].
- **Optionality.** The conventional direct pointer reference model (Figure 7.1.a) is still supported. This allows the recording of aliases to be switched on only when required (Section 7.2.2).
- **History.** Apart from the object pointer, an alias carries information about the object reference it represents. Through the relationship with other aliases, two main dimensions of object-oriented runtime behavior are captured: historical object state (Section 7.2.3) and object flow (Section 7.2.4).

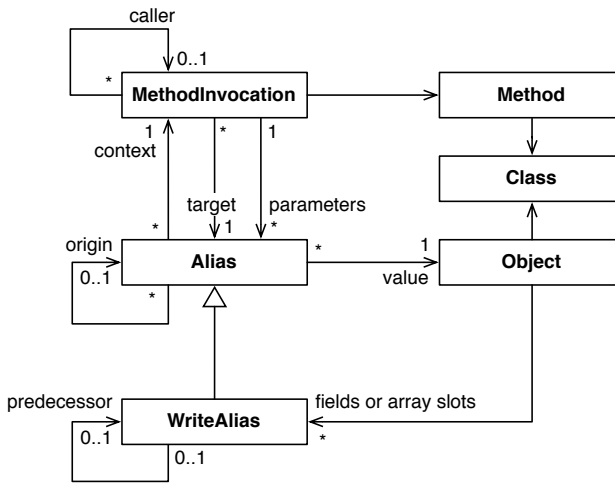

Figure 7.2: Object Flow Analysis metamodel.

Representing aliases directly as conventional objects allocated on the heap simplifies the internal object model of the virtual machine and allows us to use the standard garbage collector without needing to adapt it. For the same reason, many virtual machines represent classes and methods as internal objects.

7.2.2 Capturing Object References

In contrast to other back-in-time debugging approaches, which typically collect and store data centrally as a trace of events, aliases are part of the object model. Like events, aliases capture historical execution data, but instead of ordering them in a temporal trace they are attached to object references. For example, in the case of writing to a field the alias objects are directly pointed to from the corresponding field of the object. For each value there can exist many aliases, whereas an alias always points to exactly one value.

Illustrated in Figure 7.2, parameter aliases are referred to from the method invocation in which they replace the pointer to the actual parameter objects. The context of an alias is used to navigate to the method invocation in which the alias was created. To model the call stack, each method invocation holds onto its caller. Aliases are pushed on the operand stack of the method invocation the same way as the objects they point to would be. Exceptions are field and array write aliases, since they are referred to

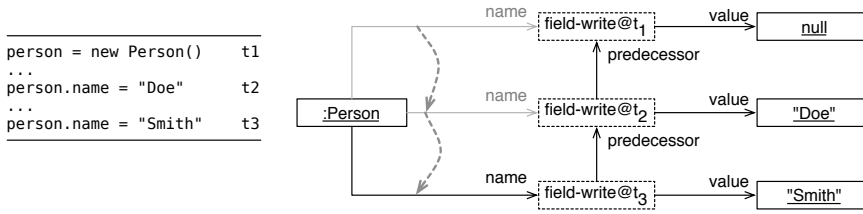

Figure 7.3: Capturing historical object state through predecessor aliases.

only from fields and array slots. When they are accessed a new read alias is created (for a complete overview of the different kinds of aliases, please see Chapter 3).

In the following two sections we detail on the use of the predecessor and origin associations between aliases in the virtual machine. These two orthogonal relationships between aliases are key to our approach as they capture object state changes and track the flow of objects.

7.2.3 Remembering Historical Object State

An important historical dimension for back-in-time debugging is how the state of an object evolves. This allows a back-in-time debugger to answer the question: *What were the previous values of a field and where in the control flow were they assigned?* More precisely, we want to capture data that allows us to later determine which value was stored in a field of an object (or at a specific index of an array) at a given point in the execution of the program.

The *predecessor* relationship in the Object Flow Analysis metamodel captures this dimension.

Figure 7.3 illustrates an example of a person object with the attribute name. When the object is allocated at the point in time t_1, the field is initially undefined. Later, at t_2, the string "Doe" is written into the field and at t_3 it is renamed to "Smith".

In the example the field first points to the write alias of null, then to the field write alias of "Doe" and lastly to the field write alias of "Smith". Each alias keeps a reference to its predecessor, the alias that was stored in the field beforehand. In this way, the alias pointed to from a field is the head of a linked list of aliases that constitute the history of that field.

Looking into the past. To go back in time, a selected process can be put into a state in which it "sees" the system as it happened to exist at a certain point in the past. Like this, accessing a historical value of a field is automatic

```
if process.timestamp is not defined then
    return x.f
else
    alias := x.f
    while (alias.timestamp > process.timestamp and
        alias.predecessor is defined)
            alias := alias.predecessor
    return alias
end if
```

Figure 7.4: Pseudo code for the VM implementation of field access `x.f` with back-in-time capability.

because when accessing a field (or array), the historical value is returned directly — just like the current value is normally returned.

Figure 7.4 shows pseudocode for the implementation of field access in the virtual machine. In case the current process has an activated back-in-time view, the predecessor list of the currently referenced alias is traversed backwards to the alias that was present in the field at the selected point in time.

In the example of Figure 7.3, accessing person.name at timestamp t_3 directly returns the alias of the string "Smith" whereas at t_1 an alias of the undefined value is returned.

With this model, previous object state can be accessed very quickly (depending on the number of state changes of the field, which is typically a small number). Compared to other approaches, which need to reconstitute previous object state from a log or from a database, this is significantly faster.

7.2.4 Remembering the Flow of Objects

In addition to the historical object state dimension discussed above, we want to capture how objects propagate at runtime. The goal is to answer the second key question: *How was this object passed here?* This means, for any object accessible in the debugger, we want to be able to inspect all origins up until the allocation of the object. This also allows us to find out where a particular value of a variable comes from. Furthermore, we also want to track the flow of the undefined value and any primitive values. Tracking the undefined value is important as null pointer exceptions can be hard to debug.

The way we track the flow of objects is similar to that of tracking past object states discussed above. The origin relationship in the Object Flow Analysis metamodel captures all transfers of object references in a system (when recording is turned on). In other words, to capture how an object

is passed around, each alias of the object, except for the creation alias, maintains a link to the alias from which it originates.

Figure 7.5 illustrates the flow of an account object in an execution trace (represented as a tree of method invocations where the callee points to the caller). A point in Figure 7.5 represents an alias. An arrow from one alias to another shows the origin of the alias. Note that the actual flow of the object is opposite to the direction of the origin arrows. A box indicates a method invocation (stack frame). Each method invocation links to its calling invocation.

Each alias is created in the context of the method invocation in which the object reference becomes visible. This means that for return values this is the calling method rather than the returning method. The parameter aliases are created at the callee site.

By means of the origin link of an alias we can track back how an object was passed to a method invocation in which a failure occurred. This helps one to understand how and why a possibly incorrect object reference has been propagated — even and especially if its flow spans the whole program execution and goes through fields and arrays.

Introspecting object flows. Aliases are completely invisible at the application level because they forward all messages to the actual objects. Therefore, we have to provide other means to access object flow information than to send messages to an alias instance. We employ the concept of Mirrors [BRAC 04] to introspect aliases. For each object reference that is represented by an alias instance, a mirror can be obtained through a primitive. A mirror is an object that provides an interface to access for example the origin and the predecessor of an alias, which in both cases returns a new mirror of the corresponding alias. In the same way we can access the method invocation context of an alias to get information about where in the control flow the alias was created. Our prototype implementation of the enhanced debugger uses this mechanism to navigate backwards in time.

7.2.5 The Effect of Garbage Collection

The upper part of Figure 7.6 illustrates the same execution trace and object flow illustrated by Figure 7.5. The currently active call stack is highlighted on the right side. The method invocations and the aliases are nodes in the memory graph, whereas the caller and origin arrows are the directed edges in this graph. The effect of a garbage collection is illustrated in the lower part of Figure 7.6.

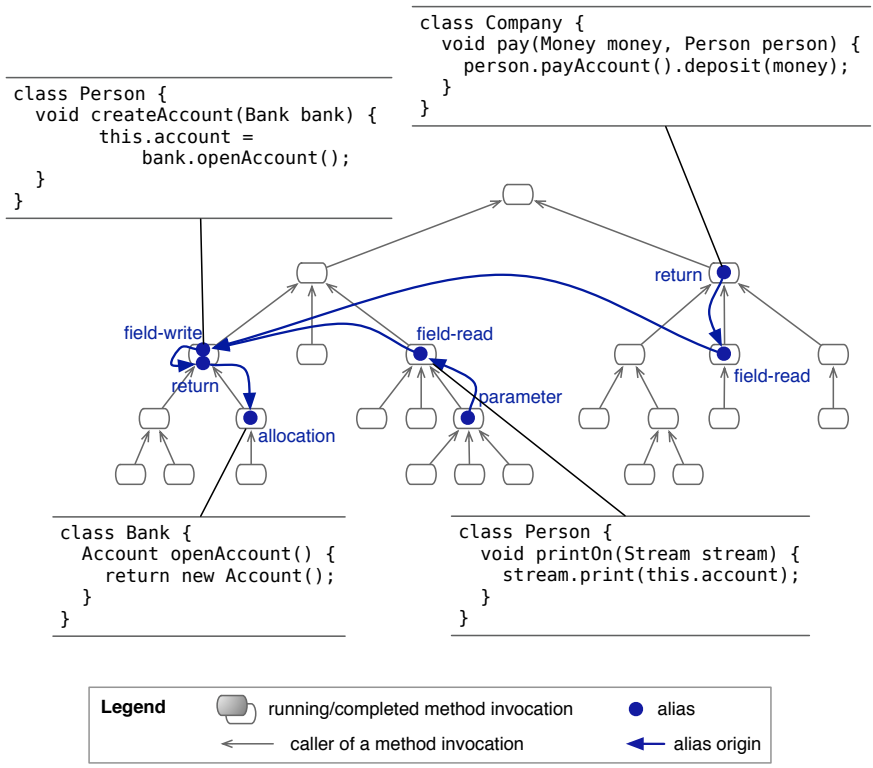

Figure 7.5: Flow of an Account instance through an execution tree.

In this example, the following objects survive the garbage collection:

- The current call stack is preserved since the active invocation (1) is considered a root object.
- The return alias (2) of the invocation payAccount() is not deleted since it is referred to from the operand stack of the active invocation pay().
- The invocation of payAccount() (3) is also preserved as it is the context of a field read alias and the field read alias is the origin of the return alias (2).
- The corresponding field read alias originates in a much earlier branch of the call tree where the account instance was written to a field after it was returned from the method invocation openAccount() (4).

The branch of the object flow (the field read and parameter aliases in the middle of the execution tree) does not survive the garbage collection. The

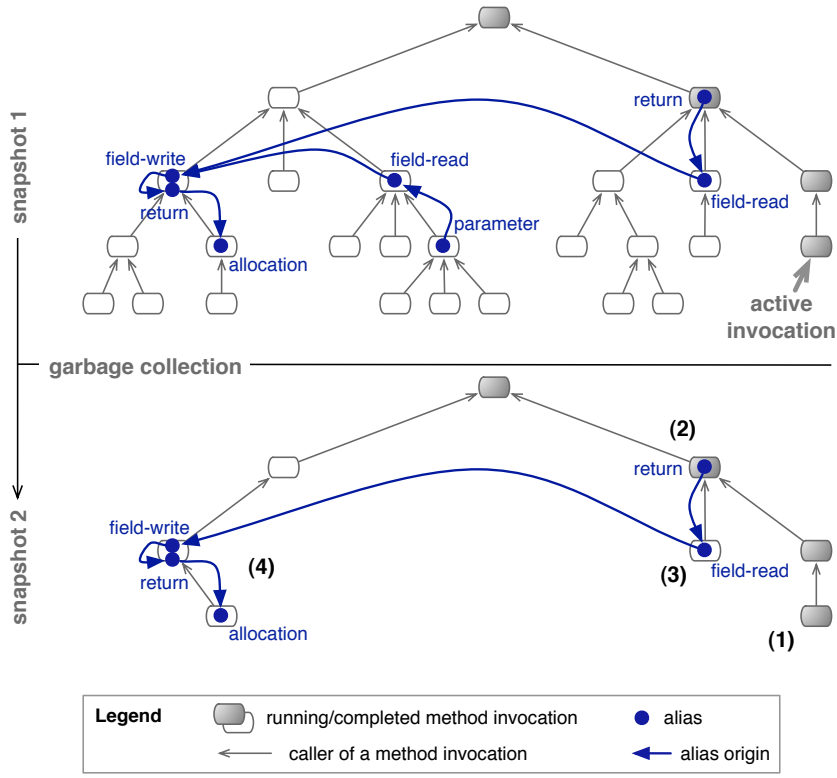

Figure 7.6: Flow of an object through an execution tree and the effect of garbage collection.

reason is that no other object flow exists that would make a connection to the alias and invocation sub-graph. Also, many method invocations do not survive as they are not subject to relevant object flows and as they are not in the caller chain of a relevant invocation.

Figure 7.7 shows memory statistics from the execution of the Squeak bytecode compiler. In regular intervals we measured how many aliases have been allocated in total (solid line) and how many of those aliases still exist in memory (dashed line) over time. The effect of the garbage collection over the whole execution is a reduction of data by 70%. Both statistics in Figure 7.7 show the same run of the compiler, but with different garbage collector settings. On the left side, there are fewer GC cycles. For instance between 50ms and 120ms there is no GC activity and therefore both lines increase at the same rate. On the right side, between almost all sample steps there is a GC cycle.

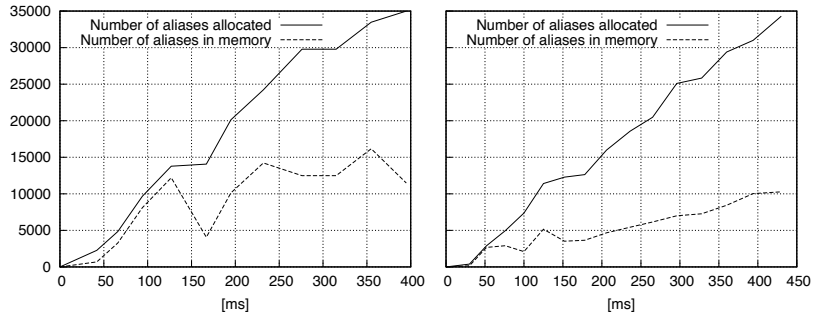

Figure 7.7: Garbage collection discards 70% of the aliases in a run of the compiler. The right side shows the same execution but with a flatter curve due to more GC activity.

7.3 Implementation

We have extended the Smalltalk Squeak VM [INGA 97] with the recording capability and representation of the execution history in the object model as described in Section 7.2. The majority of the Squeak VM is implemented in a subset of Squeak Smalltalk, named Slang. The Slang source code is then translated to C to compile and link with the low-level, platform-specific C code. The Squeak VM implementation closely follows the specification given in the Smalltalk-80 Blue Book [GOLD 83], except for the object memory format. Like most modern virtual machines, Squeak implements references as direct pointers rather than using an object table.

We implemented aliases as real objects of a new class Alias that has the fields value, context, origin and predecessor as illustrated in Figure 7.1 and Figure 7.2. In addition, Alias has two integer fields to encode the timestamp, the current program counter (for the precise location of the alias creation in the method), and the kind of the alias (one of the eight kinds described in the Object Flow Analysis metamodel in Section 3.1).

Representing aliases as ordinary objects in memory has the advantage of simplifying the implementation. Most importantly, no changes to the object memory layout and to the garbage collector are necessary. The two main changes to the virtual machine are to allocate and initialize aliases, and to forward message sends to the actual target object in case the object is aliased. Aliases are created in the bytecode routines (*e.g.*, read and write aliases), on method invocation (parameter and return aliases), and when instantiating a class. There exist a few exceptional classes for which no aliases are created to

simplify the implementation where aliasing is not important. Those classes are Process, Semaphore, MethodContext, BlockContext, and CompiledMethod.

Since method invocations are already represented as objects in Squeak (instances of the class MethodContext), to implement the model as illustrated in Figure 7.2, no further changes are required.

To optimize performance and space we implement Alias as a compact class. That is, the object header of its instances consists of only a single 32-bit word and contains the index of its class in the compact classes table. This spares one word per alias instance, but more importantly, it allows the virtual machine to check whether an object is an alias or a real object by looking at the object header alone. The efficiency of this check is especially important because not every object reference is represented as an alias and hence this check has to be performed very frequently.

To generate the different kinds of aliases when tracing is enabled, we extend the instantiation primitive, the fetch and store bytecode routines, and the message send routines. These extensions conform to the specification of Object Flow Analysis given in Section 3.2. The main differences of the virtual machine implementation to the formal specification are:

- Aliases store additional information like a timestamp and their kind encoded in integer values. Since the virtual machine is translated to C, which does not support inheritance, we only use one class to represent aliases, unlike the conceptual alias class hierarchy discussed in Chapter 3.
- There is no distinction between the heap and the alias store. In our implementation alias instances are allocated in the same memory space like normal objects, which is critical to employ the garbage collector.
- While in the specification each object reference is represented by an alias, in our implementation aliases are optional. This enables the scoping to only record the execution of a desired process or of selected parts of the code.

Small modifications have to be applied to many other primitives and bytecode routines than the ones mentioned above in case they operate on the actual object rather than on the alias. For instance in arithmetic operations and the jump bytecode routines, the target and the values popped from the operand stack need to be unwrapped if they are aliases.

Overall, about 200 methods of the Slang implementation are modified or created. In comparison, the core of the virtual machine is implemented with about 750 methods (not counting platform specific code directly written in C and plugin code). Half of our changes are necessary due to the need of

unwrapping aliases. Other parts of the virtual machine, for instance the memory format, method lookup, and the garbage collector, do not require any modifications.

At the application level, only very few extensions in system classes are required to support recording, to allow the user to control recording, and to introspect the execution history. First, the class Alias has to be loaded. It implements no methods and cannot be instantiated by the user. Second, the class Process is extended to allow the user to control recording at runtime. We add a field to Process that specifies the recording mode as well as the timestamp of its back-in-time point of view. At runtime, the behavior of the virtual machine then depends on these settings of the active process.

In addition, we implement the class AliasMirror, which can be loaded to introspect the execution history. Mirrors are used by the graphical debugger, which we modified to be capable of moving backwards in time and to navigate backwards in the flow of objects. The debugger accesses information about the flow and history of an object by requesting a mirror for the reference through which the object is made visible in the selected method invocation. There is no need to traverse the heap or perform a lookup in a trace to get the flow or history of an object, since this information is available through direct object references. A mirror on an object reference is created by the virtual machine through a primitive call. Using a mirror on an alias, the fields of the alias can be accessed. This behavior is implemented with a set of primitives in the virtual machine because any direct access to the alias would be performed on the actual object rather than on the alias instance.

7.4 Performance Evaluation

In this section we evaluate our implementation from the point of view of the execution overhead (Section 7.4.1) and of the memory usage (Section 7.4.2). All experiments were performed on a MacBook Pro, 2.4GHz Intel Core 2 Duo, 4GB RAM, with Mac OS X 10.5.2.

7.4.1 Execution Overhead

Setup. To evaluate the execution overhead, we compare the performance of the modified Squeak virtual machine to the original virtual machine by means of several standard benchmarks. As a reference, we first executed the benchmarks in an original Squeak virtual machine (version 3.9-10), which we compiled using gcc 4.0.1. Then, we executed the same benchmarks using our modified virtual machine, which had been compiled under identical conditions. First, we took the benchmarks with the recording of historical

data being turned off, and second with recording turned on. For each of the three cases the five benchmarks were executed 30 times, and before each execution we forced a full heap garbage collect.

Overview. The results of this comparison are shown in Table 7.1 and Table 7.2. The first three columns show the results of the benchmarks executed on our modified virtual machine without historical data recording, that is, no aliases are created. The remaining columns to the right show the results obtained from running the benchmarks with recording turned on. These overheads include the time that is needed to allocate and initialize alias instances, the additional time to forward message sends from aliases to normal objects and the additional time spent in the garbage collector.

The most important numbers are shown in the two Δ columns, which indicate the execution overhead of the benchmarks compared to the reference run of the unmodified standard virtual machine. The column *time* is the runtime of the benchmark and *%GC* indicates how much of this time is consumed by the garbage collector. The last two columns of the table show how many alias objects respectively method invocation objects are created (k indicates that both figures are given in 1000 objects). The figures in Table 7.1 and Table 7.2 are computed using the arithmetic mean of the 30 runs of each benchmark.

	Recording OFF		
Benchmark	Δ	time[s]	%GC
Tiny benchmark (bytecodes)	**1.02**	1.03	0.0
Tiny benchmark (sends)	**1.20**	1.39	0.0
STones80 (low-level)	**1.12**	0.51	5.6
STones80 (medium-level)	**1.27**	0.38	0.4
Squeak macro benchmark	**1.16**	0.38	2.0
Average	**1.15**	0.74	1.6

Table 7.1: In comparison with the original VM, the execution overhead of the modified VM averaged 15% when recording is disabled (see column Δ).

Benchmarks. We used five different benchmarks[1]. The first two rows show the results of two *Tiny benchmarks*, which measure how many bytecodes and message sends can be executed per second. The bytecode benchmark is based on a bytecode-heavy implementation of the "Sieve of Eratosthenes" whereas the message send benchmark is based on a send-heavy recursive calculation of Fibonacci numbers. The second and third rows

[1]The Tiny benchmarks can be found in the standard Squeak distribution. The STones80 and the Squeak macro benchmarks can be found in the Benchmarks package on http://map.squeak.org/

		Recording ON			
Benchmark	Δ	time[s]	%GC	aliases	methods
Tiny benchmark (bytecodes)	**2.26**	2.29	16.1	13773 k	8 k
Tiny benchmark (sends)	**2.06**	2.39	7.1	7881 k	11406 k
STones80 (low-level)	**1.53**	0.70	8.4	21600 k	4960 k
STones80 (medium-level)	**6.43**	1.91	46.0	59478 k	17077 k
Squeak macro benchmark 2.0	**6.91**	2.23	60.7	5532 k	669 k
Average	**3.84**	1.91	27.6	21653 k	6824 k

Table 7.2: In comparison with the original VM, the average slowdown when recording is enabled is 3.84 (see column Δ).

show the results of the *STones80 benchmarks*, which are available for many different Smalltalk dialects. Whereas the low-level version mainly involves arithmetic operations, array operations, and object allocation, the medium-level version also performs recursive calls, collection and stream operations. The last row in Table 7.1 and Table 7.2 shows the results of the *Squeak macro benchmark*, which measures decompiling and then re-compiling methods.

Discussion. The results of the benchmarks taken with disabled recording averaged 15%. These numbers are a good indication of the performance penalty caused by our virtual machine modifications. When recording is turned off, no aliases are allocated and hence no message sends have to be forwarded and no additional time is spent in the garbage collector. Interestingly, the overhead of the Tiny bytecodes benchmark is very low with an overhead of only 2%, while the overheads of the other benchmarks average between 12% and 27%. To find out whether this difference is significant or whether our modifications have any measurable influence on the performance of the bytecode benchmark, we performed the following statistical analysis.

We formulate the null hypothesis H_0 that the average runtime of the Tiny bytecode benchmark is not slower when executed on our modified VM (M) compared to the original VM (O), formally: $\mu_O \geq \mu_M$. The alternative hypothesis H_1 postulates that the average runtime of the benchmark is slower when being executed on our modified VM compared to the original VM, formally: $\mu_O < \mu_M$.

To test the hypotheses we apply the independent one-sided two-sample t-test [KANJ 99] with an α value of 1% and 58 degrees of freedom. The variance requirement is fulfilled and both data sets are normally distributed (verified with the Kolmogorov-Smirnov test). We calculated a t value of -16, which means that we can clearly reject the null hypothesis H_0 and accept the alternative hypothesis H_1 (the t distribution tells us that the probability that $t \leq -2.4$ is 1%). Therefore, we can conclude that the 2% slowdown

of this benchmark is due to our modifications of the VM. Using the same method, we can draw the analogous conclusion for all other benchmarks. This result is not surprising as those runtimes are clearly distinct from the reference runtimes.

In the case of recording switched on, particularly noticeable are the big differences between the overheads of the first three low-level benchmarks compared to the other higher-level benchmarks. The overheads of the medium-level STones80 benchmark (factor 6.43) and the Squeak macro benchmark (factor 6.91) are more than three times as big as the overheads of the low-level benchmarks. One reason for this is that those benchmarks spend a significant percentage of their runtime in the garbage collector (46.0% and 60.7%). The high pressure on the garbage collector cannot be explained entirely by the higher rate by which alias and method invocation objects are created (31 million aliases/s for medium-level STones80 and 2.5 million for the macro benchmark). For instance, the low-level STones80 benchmark produces 30.7 million aliases/s but incurs a relatively low overhead. Rather, it is likely that the high pressure is due to the different memory usage characteristics, *e.g.*, how fast aliases can be garbage collected.

Summary. These benchmarks suggest that a significant overhead incurs because of the additional pressure on the garbage collector, which depends on the characteristics of memory usage. (Memory usage characteristics are further discussed in Section 7.4.2.) In turn, instantiating and initializing aliases seems to contribute not as much as the garbage collector to the overhead.

Without much optimization effort the overhead of our implementation when recording is switched off is just 15%. This suggests that with more aggressive performance optimizations (such as using different sets of bytecode routines when tracing is disabled) a virtual machine enhancement for capturing execution history could potentially be incorporated into a standard distribution. This would allow users to switch recording on and off as required without needing to recompile code and restart the application with a different virtual machine.

7.4.2 Memory Usage

To further evaluate the practicality of our approach, we also investigated the characteristics of larger applications with respect to the amount of memory consumed. Of particular interest is whether the retained historical data increases steadily over time, or whether the amount of data is bounded by an upper value.

We expected this characteristic to depend on the type of application. For example, in applications with persistent objects that undergo many

state changes, this is obviously not possible as all previous object states are retained until the objects themselves are garbage collected. In contrast, in applications where objects are used only temporarily it is possible that long running programs can be recorded without running out of memory.

To study different types of memory usages we chose the following three programs:

- *A program that allocates a large number of temporary objects that get garbage collected after some time.* We selected the Squeak bytecode compiler which we ran on 1000 classes. We expected that the history of objects generated to represent tokens, AST nodes and intermediate representation objects can be garbage collected after the bytecode of a class has been successfully emitted.
- *A program with a stable number of objects that undergo a large number of state changes.* We selected a gas tank simulator shipped as a Squeak demo. Each molecule in the tank is represented by an object and on each GUI update the position of the molecules, their velocities and directions are recalculated and changed. Since all previous positions of the molecules are remembered, we expected a lower effect of the garbage collector in comparison to the bytecode compiler.
- *A program with an existing object graph that is heavily accessed and modified.* We chose a commercial web content management system (CMS). The history of modifications of the object model and the behavior leading to it cannot be discarded after some time because the object model of the CMS is completely kept in memory.

Bytecode Compiler

Figure 7.8 shows virtual machine statistics taken from sampling the execution of the Squeak bytecode compiler. We compiled 1000 classes from the Squeak Smalltalk system which took 652 seconds when recording was turned on (compared to 168s when recording was disabled). In total the execution produced more than 2 billion aliases (solid line in Figure 7.8) and 443 million method invocations (not shown). On average, 3 million aliases are created per second. The actual number of aliases in memory was relatively low at an average of 2.9 million (notice the different scales of the left and right Y-axes). The maximum amount of memory allocated by the virtual machine was 317MB.

The temporal development of the number of regular objects (excluding alias objects) is similar to one of the number of aliases. The recurring pattern of growth and decline is caused by incremental and full garbage collect cycles.

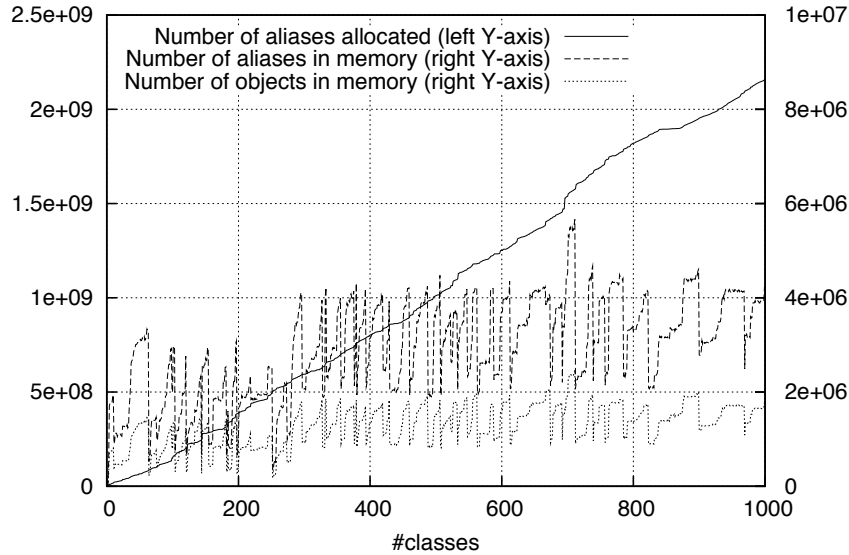

Figure 7.8: Compiling 1000 classes (X-axis) produces more than 2 billion aliases, however, the number of aliases in memory stays below 6 million. Please note that because of the large differences between the number of allocated aliases (solid line) and the aliases and objects in memory (dashed and dotted lines), we use two scales: one for the solid line to the left and one for the dashed and dotted lines to the right.

The analysis of this application showed the expected behavior. The historical data kept in memory does not grow without limit because the compiling history of a class is discarded after the bytecode has been generated and emitted.

Gas tank simulation

Figure 7.9 shows the analysis of the following usage scenario of the gas tank simulation. First we started one instance of the simulator and paused it after the sample step #5. Then we started a second simulator with twice the number of molecules. The higher rate of aliases allocated from this time on is reflected in Figure 7.9. At step #10 we quit the second simulator and resumed the first one.

Quitting the second simulator has a big effect on the number of aliases retained in memory (see decline between steps #10 and #11). Since the objects of the second simulator are not accessible anymore, the remaining

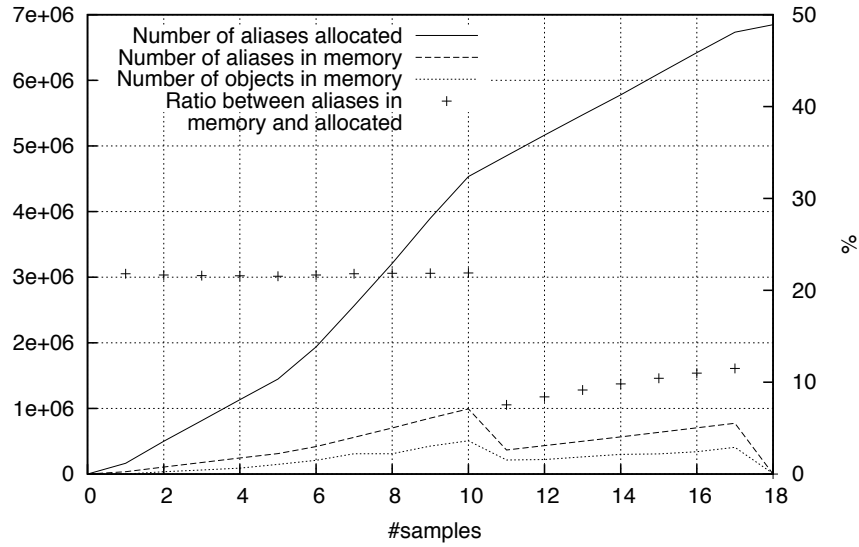

Figure 7.9: Analysis of a gas tank simulator shows that 22% of the aliases allocated are retained in memory (19 samples with an interval of 3s each).

execution history is garbage collected. The same happens after quitting the simulator at the end of the analysis, where the number of aliases in memory drops to zero.

A striking difference to the case of the bytecode compiler is that the number of aliases in memory grows with respect to the number of aliases allocated over time. As Figure 7.9 shows, the ratio is constantly 22% in the first half of the analysis up until the event where the second simulator instance was quit. This means, that for this application 78% of the aliases are garbage collected but the rest adds up in memory and eventually the virtual machine will run out of memory for long runs.

The execution history retained by our approach allows one to revert the state of objects that are currently accessible (or that are accessible through a past field reference of an accessible object). In case of the simulator this means that we can move back in time as long as the simulator user interface has not been closed. For instance we can set the point of view of its GUI process to a past point in time. This has the effect that the molecules are displayed where they were positioned at that time, and that also all settings of the simulator are reverted to their previous states. To find out how a position was calculated, we can follow back the flow of the corresponding point object to where it was allocated.

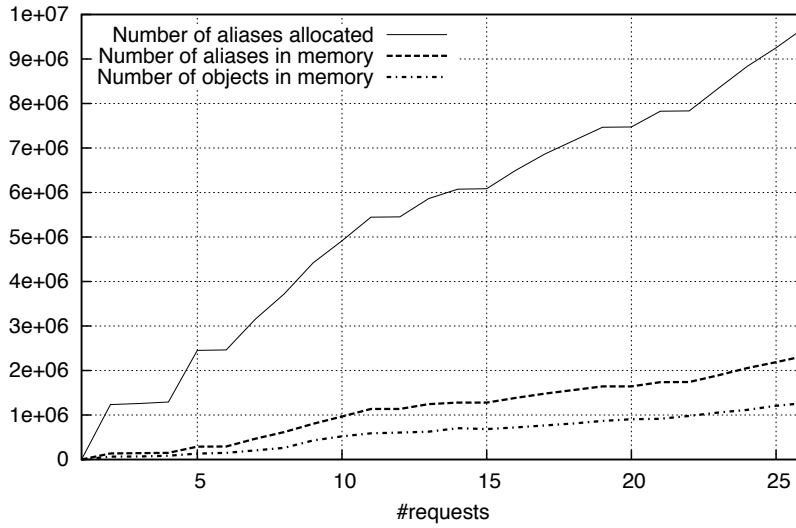

Figure 7.10: Analysis of a user session in a Content Management System. After 26 requests, 24% of the allocated aliases are still in memory.

Content Management System

Figure 7.10 illustrates an analysis of a user session in Cmsbox[2], a commercial web content management system. The session consists of 26 user actions such as login, editing content, drag and drop, copy and paste elements, publish page, etc.

We chose Cmsbox as a case study, because it stores all objects in memory rather than in a database, which makes it a worst case scenario for our approach. Indeed, as the figure shows, the aliases do increase steadily over time, the main reason being that more objects are added and retained in the model and in memory.

This experiment shows the limits of our approach. However, as described in Section 7.5.1 we can limit the effect of this phenomenon through selective recording of aliases.

[2] http://www.cmsbox.com/

7.5 Discussion

Memory consumption and performance can be further tuned by adjusting the level of detail of information gathered. We now look at several ways this can be done, and we discuss difficulties, limitations and potential optimizations of our implementation.

7.5.1 Capturing and Remembering Less Data

Depending on the usage of the back-in-time debugger, for instance in a testing or production environment, it can be necessary to further decrease memory consumption and lower the execution overhead. A common solution to reduce the amount of data recorded is to not instrument all code, for instance by excluding libraries and framework code. The effect is that in the code that is out of scope, no side effects are captured and the links of objects being passed through this code are lost.

We experimented with an alternative approach that is not based on structural scoping but on tuning how fast recorded data is discarded. In particular, in our implementation the user can change the behavior of the virtual machine by (a) disabling tracking of predecessor aliases, (b) disabling tacking of origin aliases, and (c) selecting which types of aliases are created. By default, all predecessor and origin aliases and all types of aliases are recorded.

For example by not tracking predecessors, we can reduce the consumed memory but still benefit from being able to inspect object flows. By means of such configurations we can provide the same functionality as the following two debugger extensions that have been proposed recently. They are specialized to a particular debugging task and hence only need to track a fraction of the whole execution history.

Reverse watchpoint, an approach proposed by Maruyama *et al.*, analyses the execution and moves the debugger to the last write access of a selected variable by re-executing the program from the beginning [MARU 03]. This technique automates the task of finding where a variable was erroneously written and then moves the debugger to that point. With our approach, finding where a variable was written means to move one step back in the object flow from a field read alias to its origin, which is the field write alias. If we want to gather exactly this information, we can disable predecessors, and restrict the recording to create only field write and array write aliases. The effect is that for each field the most recent write alias is available with the execution stack in which it was created. When writing to a field, the previous write alias of the field or array can be discarded immediately because it is not referenced as a predecessor or origin alias.

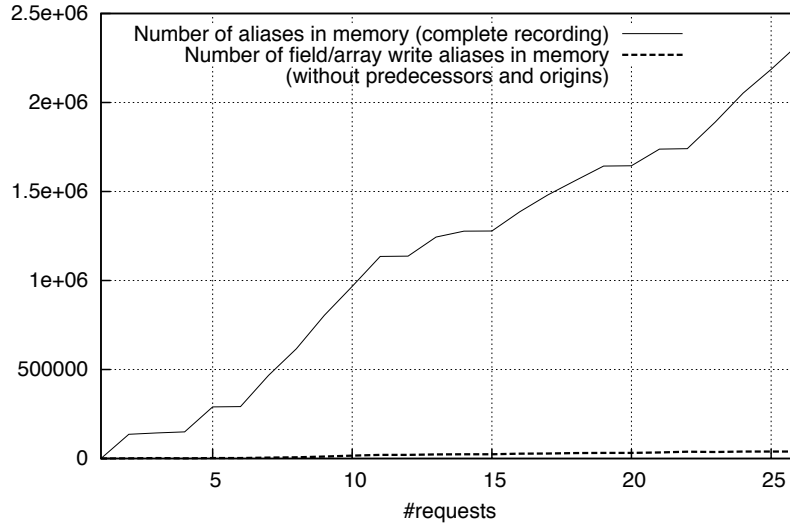

Figure 7.11: Comparison of the number of aliases retained in memory with the default configuration compared to the configuration where only the last write alias of each field and array slot is remembered (same run of the CMS as in Figure 7.10).

Our benchmarks show that with this minimal configuration, we achieve a very low execution overhead that is only insignificantly higher than the base slowdown of 15%, which the virtual machine incurs when recording is turned off completely. In comparison, Maruyama report a slowdown of their technique of about 400% [MARU 03]. Figure 7.11 shows that with this reduced configuration only a fraction of the aliases are remembered, compared to the aliases remembered using the standard configuration as illustrated in Figure 7.10.

Origin tracking of null values, proposed by Bond *et al.*, is a very efficient technique to track the method in which an undefined values originates to support debugging the well-known problem of null pointer exceptions [BOND 07]. We can do the same by only tracking aliases of type *undefined*, which again incurs a similar overhead to the configuration discussed above. In comparison, the approach of Bond *et al.* adds an overhead of only 4%. The low overhead is made possible by only tracking undefined values, which allows for "value piggybacking", a technique to store origin information directly in pointers (in this case in null pointers).

7.5.2 Remembering Control Flow Dependencies

With the default configuration our approach retains information about the flow of objects and previous states of objects. In contrast, conventional back-in-time debuggers typically record and store the complete execution history (until they run out of memory). While our approach records the same data, it discards most of it within a very short time and only remembers what is relevant for the object flow and historical states of objects that are accessible at the current point of execution. To provide enough context, with each alias, the method invocation where it is used is retained, including the execution stack with all target objects and objects passed as parameters.

Still, depending on the kind of bug, it is possible that relevant information is missing. In particular, no links between aliases exist to represent the fact that the value of a variable influences the value of another variable. For example, in the statement "if x.f then y.f = 1" there is no dependency link between the field read alias x.f and the field write alias y.f. In case the value of y.f turns out to be incorrect because of the unexpected execution of a branch, it is possible that x.f has already been discarded. However, this does not happen if x is the target object of the method or one of its parameters. In this case it is referenced from the method invocation which in turn is referenced as the context of the field write alias stored in y.f. (How exactly aliases and method invocations refer to each other is illustrated in Figure 7.2.)

We decided not to explicitly capture such control flow dependencies because we believe that the most difficult bugs in object-oriented programs are caused by subtle inconsistencies in object graphs and by the propagation of unexpected object references. However, it would be possible to extend our approach to also include dependency relationships between aliases. Each alias would need to maintain a list of other aliases it depends on, similar to the predecessor and origin relationships. The dependence information could be computed by an intra-procedural static analysis. An inter-procedural analysis would not be necessary since this dependency is indirectly captured by the link of an alias to the target of the invocation in which it is created (alias→ context→ target).

7.5.3 Limitations and Potential Optimizations

Not freed memory. Using our back-in-time debugger we noticed a couple of times that parts of the history were unexpectedly not garbage collected. The reason turned out to be that the program execution in those cases produced subtle side effects on global state. The effect is that the part of the execution history that produced the side effect is not garbage collected as long as the global state that was modified exists. We observed three cases of this problem: singletons, caches and writing to a log console (strings passed

from the application are stored in the console stream and hence retain links to the execution history where they originate). While in some cases this can be the desired behavior (in case of the cache or singleton), it may be undesired in other cases (the console). To remedy the undesired cases we can simply disable recording of the appropriate methods or classes. The real difficulty, however, is to first find the cause of such a problem. What is missing are high-level views to inspect and navigate the recorded data.

Capturing non-word size data. A limitation of our implementation is that the history of values stored as non-word data are not captured. For instance, in a byte array where four bytes are stored per word, we cannot use the approach of exchanging a value with an alias indirection because the alias pointer requires 4 bytes. Typically this is not problematic since Squeak uses non-word fields in most cases to represent internal data of objects only, such as float objects, large integer objects, or strings.

Potential optimizations of our implementation. As our benchmarks show, the slowdown is mainly caused by the additional garbage collector activity. An optimization of the garbage collector to better cope with the special characteristics of our virtual machine would improve the performance but is not straightforward to realize.

A different optimization that would also improve the performance of the virtual machine when tracing is turned off is to use different sets of bytecode routines. The current implementation uses conditionals in bytecode routines and primitives to execute code depending on the tracing state of the current thread. Implementing two sets of bytecode routines would allow the virtual machine to switch jump tables when recording is toggled.

The memory consumption of our implementation could be slightly optimized by distinguishing between field/array write aliases and the other types of aliases. Field write and array write aliases require the predecessor field to hold onto historical state, whereas the other aliases do not need this field. Using two different classes of aliases would be simple to implement and would save one word per non-historical alias instance.

7.6 Related Work in Back-In-Time Debugging

Logging-based approaches. The most common approach to implementing back-in-time debuggers has been to create a trace log of the program execution. ZStep95 is a reversible debugger for Lisp that provides animated views but does not address performance and scalability issues [LIEB 98]. Lewis proposed ODB [LEWI 03], a back-in-time debugger for Java, and

Hofer proposed Unstuck [HOFE 06], a similar proof of concept implementation for Squeak Smalltalk. Both approaches have in common that they keep the log history in memory and hence can only record and store the complete history for a short period of time. ODB allows one to set a fixed limit on the number of events and it then discards older events when the limit is reached.

A more scalable approach has recently been proposed by Pothier *et al.* [POTH 07]. Their back-in-time debugger, TOD, addresses the space problem by storing execution events in a distributed database. While this approach has the benefit that no data is lost, its drawback is that it requires extensive hardware power, which is not available for many developers today. To cope with the data generated by a CPU-intensive program, 10 database nodes in a server cluster are required. Also, the approach has a performance overhead of a factor 113 in the worst case, which is approximately the same as the one of ODB for the same benchmark [POTH 07].

In comparison, the performance of our approach is about one order of magnitude better. On the one hand, this is because our approach is implemented at the virtual machine level, whereas all previously mentioned approaches are based on bytecode instrumentation. On the other hand, as our approach stores historical data directly in the application memory, it does not require any additional logging facility to gather and store data. As a side effect, our representation of historical information is also very space efficient. For example, there is no need to assign identifiers to objects or to serialize objects since they exist in memory and can be referred to directly by pointers.

Outside of research, back-in-time debuggers have unfortunately not been widely adopted yet. An example of a commercial back-in-time debugger is Omnicore's CodeGuide[3]. It is also based on bytecode instrumentation and its execution history, which is kept in memory, is limited to the few last thousand events. An interesting aspect of CodeGuide is that only methods containing breakpoints and methods close to them in the control flow are instrumented to keep the runtime overhead low.

Related to logging-based back-in-time debugging is *query-based debugging*. In those approaches the user formulates a query in a higher-level language that is then applied to the logged data [MART 05, LENC 97, POTA 04, DUCA 06]. Queries can test complex object interrelationships and sequences of related events. Approaches exist that execute the query at runtime, which can improve performance because no history has to be stored [LENC 99]. Our approach, like other back-in-time debugging approaches, does not support querying for complex relationships in the history, but in return it incurs a much smaller execution overhead.

[3] http://www.omnicore.com/

Replay-based approaches. A different approach for implementing back-in-time debuggers is to replay the debugged program until a desired point in the past. To optimize the time required to reach a given point in the past, many approaches take periodic state snapshots, for instance Bdb [FELD 88] and Igor [BOOT 00]. The main advantage of replay-based approaches over logging-based approaches is their low performance overhead (roughly 2 times for Bdb and 4 times for Igor). The disadvantage of those kinds of approaches is that moving backwards in time can be very slow because the program has to be partly re-executed. This issue has been addressed in a recent publication by Xu *et al.* [XU 07b]. Our approach can access past object state almost instantly because it only needs to look up the appropriate alias in the predecessors chain (as described in Section 7.2.3). An open issue of replay-based approaches is that of deterministic replay, which cannot be guaranteed by all approaches if the program depends on external resources or if it is multithreaded.

The approach of taking (incremental) memory snapshots and replaying has also been used in the Leonardo virtual machine [DEME 04], a virtual machine based approach for assembly-like languages that features reversing program state. Similar to our approach, programs slow down by a factor of 6 in the worst case. However, the Leonardo virtual machine does not support inspecting object flows and it does not provide a strategy to discard data.

7.7 Summary of the Chapter

In this chapter we tackle the problem of how to make back-in-time debugging practical by (1) keeping memory consumption within reasonable bounds by only keeping track of still-relevant past data, and (2) reducing the runtime overhead by implementing recording at the virtual machine level. Our approach does not store all data, but instead it focuses on remembering the history of the objects that are still referenced in the current program state. Our solution makes use of the garbage collector to release the objects that are not referenced anymore in the program and that are not relevant anymore in the program's history.

Benchmarks have shown significant improvements over existing approaches. First, the memory consumption is confined to an upper bound limit in the best case, or grows slowly in the worst case. However, for the worst case scenario, we can configure the recording to capture and remember less data, which can lead to a dramatic decrease in memory consumption (*e.g.*, 55 times fewer aliases when just remembering the last field write alias). Second, performance is in the worst case 7 times slower than a regular execution. Furthermore, the modified virtual machine with tracing

switched off introduces only modest overhead (*e.g.*, in our benchmarks, it introduces an average of 15%) as compared with a regular one.

This chapter demonstrates that the conceptual model of program execution history offered by Object Flow Analysis is also valuable outside reverse engineering. Object Flow Analysis provides a consistent model that unifies the dimensions of reference transfer and reference structure, which are critical to debug an object-oriented program. This model is simple in its structure as its core can be represented by only one class and three associations (origin, predecessor, and context). Therefore, at the level of the virtual machine design, our approach requires no fundamental changes; especially no changes to complex components like the garbage collector are required.

Chapter 8

Conclusions

You cannot understand a cell, a rat, a brain structure, a family, a culture if you isolate it from its context. Relationship is everything.
— Jonas Salk

We set out to analyze dependencies introduced by aliasing in object-oriented systems, and we identified the gap in dynamic analysis of tracking the transfer of object references. Our approach, Object Flow Analysis, proposes a simple concept for modeling object flow: each object reference established at runtime is represented by a first-class entity, and the transfer of object references is captured by the relationship between the instances of this entity. Building on our metamodel, we have presented a conceptual framework that provides a basic structure to reason about dependencies. We have demonstrated how to instantiate this framework for three reverse engineering analyses that take different perspectives on the problem of object aliasing.

The Object Flow Analysis metamodel integrates with existing dynamic analysis models. In addition to the dimension of object reference transfer, also object reference structure is directly represented in our model, which provides a means for reconstructing arbitrary intermediate object graphs of a program execution. Furthermore, the exact location of object references in the dynamic control flow is captured, which is useful, for example to relate object flow to features and classes.

8.1 Contributions

By proposing a foundation for the analysis of object aliasing, Object Flow Analysis contributes to knowledge in the field. Object Flow Analysis extends the previous work by filling the gap of reference transfer analysis and by unifying this new analysis with the existing dynamic data and dynamic control flow analyses. Object Flow Analysis covers the missing aspects of aliasing to bring dynamic analysis in line with the object-oriented programming paradigm.

These findings have implications for the research in this field. As our proposed analyses have demonstrated, by explicitly representing object references, Object Flow Analysis provides a level of detail that opens new possibilities to analyze object-oriented program behavior.

As a validation of our approach, we have focused on reasoning about the dependencies introduced by object aliasing. We have provided anecdotal evidence of the usefulness of this analysis by proposing three applications based on it, and we have demonstrated that the applications can be concisely expressed with Object Flow Analysis. In summary, the contributions of these analyses are the following.

- The *visualization of object flow* provides a new perspective on the design of a system by investigating how objects flow between classes at runtime. This novel view is complementary to existing approaches that focus on the perspective of message passing (*e.g.*, using UML sequence diagrams). Our approach exposes indirect dependencies and provides insights into the conceptual flow of information in a system, which is not visible from a message passing point of view.
- The *analysis of runtime feature dependencies* provides a definition and detection strategy that takes object aliasing into account and hence is more precise than earlier approaches. The additional dependencies that we uncover are precisely the indirect feature dependencies that can be problematic during maintenance.
- The *Test Blueprint*, which is based on an analysis of object flow in execution traces, guides the developer when writing unit tests for legacy software. Our approach addresses the problem of writing unit tests for an unknown system, which is difficult because of the gap between the static structure and actual runtime behavior of an object-oriented program.

Compared to existing dynamic analysis techniques, Object Flow Analysis requires a more challenging runtime analysis because it depends on not directly observable data, like the origin of object references. To this end, we provide a formal specification for tracking object flow in a running program.

Moreover, we have demonstrated that Object Flow Analysis extends beyond the traditional application in reverse engineering. By leveraging object references to first-class objects in virtual machines, we propose an approach that features significant improvements on the memory explosion and performance problems of back-in-time debugging. The evaluation of this virtual machine implementation shows that the tracking and representation of object references as proposed by Object Flow Analysis is an effective way to capture program execution history.

8.2 Future work

Detecting dependencies between other program abstractions. The three presented analyses are based on detecting dependencies between classes and packages, features, and parts of the control flow. Object Flow Analysis could be applied to detect dependencies between other abstractions, for example between objects to detect data sharing and reference transfer. Similar, Object Flow Analysis could help to analyze concurrent programs by detecting how objects are shared and transferred between different threads at runtime. A critical situation to detect is the aliasing of an object by two threads that concurrently modify its state.

Combining reference transfer and reference structure analysis. Our visualization of object flow between classes and packages provides a high-level view of the application. The presented analysis treats all objects equal. Therefore, a better result could be obtained if we manage to distinguish different kinds of objects based on how objects are passed around. For example, one could distinguish objects that stay in a stable relationship to each other from objects that are passed around between these objects. Essentially, this would mean to combine the knowledge about reference transfer with the knowledge about reference structure. So far, our reverse engineering analyses have only exploited object reference transfer.

Simulation of garbage collection. In our current metamodel, one cannot directly find out when an object was garbage collected. Some approaches capture this data at runtime, but with our analysis technique, this is not possible because the introduction of aliases delays the garbage collection of objects. However, a possible solution is to simulate garbage collection on the model *post mortem*. This would not be very difficult, even for arbitrary points in the program execution, because all relevant data, like the history of fields and the execution

stack with target and parameters, are available. The only additional data to capture is the set of objects that the garbage collector considers to be the roots of the object graph.

Control flow dependencies. The Object Flow Analysis metamodel does not capture control flow dependencies between object references. Let us consider the statement: `if x.f then y.f = 1`. In this statement the write alias of `y.f` depends on the read alias of `x.f`. These two aliases are not directly related in our metamodel and hence, in our back-in-time debugger, it is possible that the boolean obtained from reading the field `x.f` is discarded even though the read alias `y.f` is still relevant. Such a relationship between aliases could be captured and represented similar to the origin and predecessor relationships. Concerning our back-in-time debugging approach, it would be interesting to evaluate how this additional association influences its memory characteristics.

Varying the level of detail of the recorded execution history. To further reduce the memory usage and execution overhead of the back-in-time debugger, we can provide more control for adjusting the level of detail depending on the static structure. For instance, we can gather more data in code that is young and hence is more likely to have defects. Also an interesting extension would be to let the virtual machine autonomously decide what data to record. For example, it could increase the recording detail when an error is detected for the followup runs of the same or of related code, or it could decrease the recording detail when memory gets low.

Towards self-analyzing systems. A software system should constantly analyze itself to monitor emergent properties, such as performance degradation, memory leaks, and shifts in how the system is used. By enabling dynamic analysis on live systems, innovative debugging and analysis techniques come within reach. Our back-in-time debugging approach is a step in this direction. For live systems, however, the performance overhead would still need to be significantly decreased. Our vision is that future virtual machines, which are becoming the new platforms for programming languages, will not only provide an automatic memory management but also capabilities to record and reason about its runtime behavior.

Appendix A

The Object Flow Debugger

A.1 Introduction

This appendix gives instructions to install and use the object-flow-aware back-in-time debugging system. This system consists of the Object Flow VM, which is an extended Squeak virtual machine and of support code that provides the functionality to introspect execution history from the debugger frontend. A rich user interface to navigate the execution history is provided by the *Compass* debugger [FIER 09].

A.2 Installation

A.2.1 Downloading the Compiled VM and Demo Image

The easiest way to obtain a ready-to-run version is to download the virtual machine and demo image from the following website:

http://scg.iam.unibe.ch/Research/ObjectFlow/

The download is a zip archive that contains the compiled VM for the appropriate platform (currently available are VMs for Mac OS X Intel and Ubuntu Linux) and a prepared image. At the time of this writing the provided pre-compiled VMs have been tested on Mac OS X 10.5.5 for the Intel processor, and Ubuntu 8.04 (Hardy Heron).

A.2.2 Preparing your Image

For demonstration purpose, the provided image is sufficient, but if you like to use the debugger for one of your applications, the following installation is required. The Object Flow Debugger requires support code in the image (for example, the definition of the class Alias and the extension of the class Process). To make your own image ready to be run on the Object Flow VM do the following:

1. After backing up your image, start it up using a standard Squeak VM.
2. In the Monticello browser add the following SqueakSource repository (click the button *+Repository* and select *HTTP* as the repository type).

   ```
   MCHttpRepository
       location: 'http://www.squeaksource.com/FlyingObjects'
       user: ''
       password: ''
   ```

   Open the newly created repository and load the latest version of the package FlyingObjects. While loading, the system twice warns the used because the class Process is modified ("Process should not be redefined."). Confirm the changes by clicking the button *Proceed*. After the package is loaded, save and quit the image.
3. Now start the saved image using the Object Flow VM. For example using the VM for Mac OS X, you can do so by navigating to the downloaded directory and then execute ./vm/squeak –quartz /pathTo/yourImage.image.
4. Load the package FlyingObjectsUI from the same repository as in step 2. Save your image to be able to go back to this version whenever needed.
5. To test your installation, run the unit tests in the package FlyingObjects–Tests (in the world menu click *Test Runner* and in the Test Runner's package list select *FlyingObjects-Tests*). All tests are expected to pass.

A.2.3 Installing the Compass Debugging Frontend

If you have correctly installed the Object Flow VM along with its support packages as explained in the previous section, you should now be able to install Compass using the following procedure.

1. Install GraphViz on your system if it is not already installed (Compass uses GraphViz as a graph layouting engine). You can get it from http://www.graphviz.org/.

2. Add the following SqueakSource repository to the Monticello browser.

```
MCHttpRepository
    location: 'http://www.squeaksource.com/OmniCompass'
    user: ''
    password: ''
```

3. From this repository load the package named CompassInstaller.
4. Execute the following script, which loads all required packages.

```
CompassInstaller bootstrap
```

A.3 Debugging with Compass

This section provides a quick guide to get started with Compass.

A.3.1 Starting the Debugger

There exist two ways of recording a program execution and starting the debugger.

The first way of recording data is to use the flyDuring: method implemented in the class Object. For instance, to trace the bank account example, execute the following code:

```
self flyDuring: [ BAAccount example ]
```

The execution of the block is recorded. Now the Compass debugging interface can be started by executing

```
CompassDebugger start
```

The debugger will then show the recorded data. When closing the debugger, the traced data gets deleted. If an error occurs while tracing the code, as usual a small debugger window appears. In addition to the default buttons the new button labelled 'Compass' opens the Compass debugger at the location where the error occurred.

The second way to debug is to use unit tests. We extended SUnit to re-run a failed test and record its execution before the failure is shown in the debugger. By re-running a test, as usual the small debugger window pops up, and as discussed above the Compass debugger can be started by clicking the 'Compass' button.

A.3.2 Using the Debugger

Figure A.1 illustrates the Compass debugger user interface. This section gives an overview of the different views and actions provided by Compass.

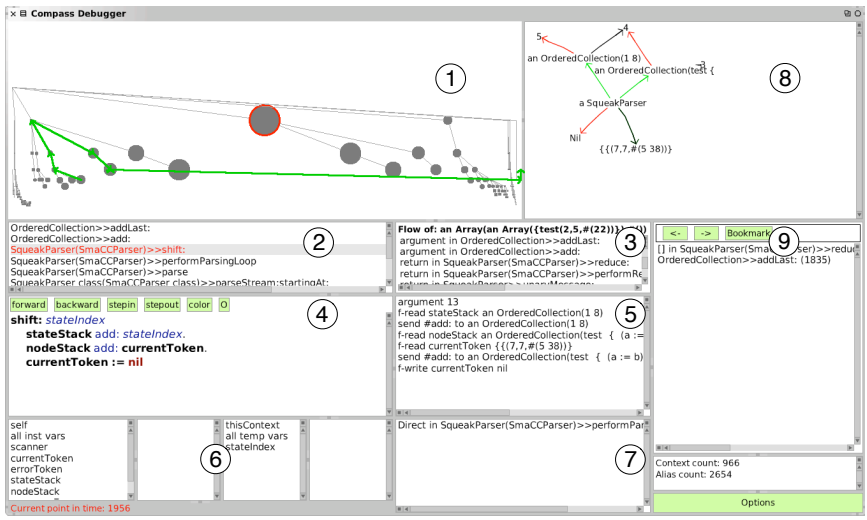

Figure A.1: Compass debugger frontend.

1. Execution trace. This view shows the execution trace as a tree in which nodes represent executed methods and block closures. Lines represent the caller relationship from top to bottom right. Nodes are ordered from left to right by the start timestamp of their execution and from top to bottom by their depth on the call stack. The trace can be navigated by clicking on the circles. The thick green arrows represent the flow of the object that was selected in one of the other views (see below).

2. Execution stack. This view shows the execution stack as it existed at the time when the selected method execution was started.

3. Object flow. This panel shows the flow of a selected object. The list contains the transfers of a reference of this object (*e.g.*, argument, return, field write, field read, etc.). This allows one to backtrack the flow of the object to find out how the object was passed into this method. The flow given by this list is the same as the one shown graphically in the execution trace (1). By selecting a reference transfer from the list, the focus of the debugger changes to the method execution in which this transfer took place.

4. Source code. This is the source code of the method of the selected method execution.

5. Executed program statements. This list shows the reference transfers (aliases) and method sends that occurred during the execution of the selected method or block execution. When an item is selected the corresponding source code statement in the source code pane (4) is highlighted. Additionally, important actions can be executed from the context menu (right-click on an item). The available actions depend on whether an alias or a message send is selected. The most important action is to show the flow of an object. When choosing this action, the flow is shown in the previously described object flow pane (3) and it is drawn in the execution trace (1). You can also choose to explore the forward flow, which brings up a window that shows a tree of how the object was transferred starting at the current selection.

6. Variables. These four panes are the same as in the original debugger. They allow one to inspect the fields of the receiver and of local variables of the selected execution context with respect to the point in time of the current focus. By right-clicking on a variable, similar actions can triggered as in pane (5), *e.g.*, selecting an object to highlight its object flow.

7. Dependencies. This list shows the control flow dependencies of the currently selected method execution (that is, the list of control flow statements present in the current call stack on which the selected method execution depends). By clicking on a dependency, the debugger jumps directly to the method execution and selects the control flow statement in the source code pane (4).

8. Side effects graph. The side effects graph summarizes the side effects that the execution of the currently selected method and all transitively called methods produced. A red arrow between two objects indicates a field or array slot update. The red arrow points from the updated object to the newly assigned object. To support the understanding of this graph, the following additional information is provided to show the connection between the different objects in the graph. A black arrow indicates the previous value of a modified field and a green arrow indicates a field or array read event (dereference). By right-clicking on an object, a menu with a list of the new field values comes up. By selecting a value, the debugger jumps to the location where the object was written into the field.

9. Navigation history. Like in a web browser, the navigation history can be used to go step by step back- and forward. In our case, the steps are the

context switches (changes of focus in the Compass user interface). If the context is changed, by clicking on the back button you get to the previously selected context. Also bookmarking is supported to be able to quickly jump to bookmarked locations in the execution history.

A.4 Miscellaneous

A.4.1 Using Alternative Tracing Policies

The Object Flow VM supports tracing policies, which define what data to recorded. This allows one to control the amount of data gathered to reduce memory consumption and execution overhead. As an example we have predefined the policy *reverse watchpoint*, which simulates an approach with the same name [MARU 03] (see also discussion in Chapter 7). To use this policy, execute the code to be analyzed as follows.

```
self
    flyDuring: [...code to execute...]
    with: TracingPolicy reverseWatchpoint.
```

This policy is very restrictive (and hence exhibits a low execution overhead) as it only tracks field write aliases but it does not maintain the origin and predecessor relationship. The field write aliases that are recorded and that are still available at the point in time a program crashes reveal in which execution context the current value of a field has been assigned.

Defining custom policies is straightforward. For an example, see the method TracingPolicy>>reverseWatchpoint.

A.4.2 Obtaining Memory Usage Statistics

To obtain the default statistics provided by the VM about memory consumption and GC activity, print:

```
SmalltalkImage current vmStatisticsReportString
```

In addition, with the support code of the OFVM a dedicated graphical memory monitor is shipped, which shows the total number of aliases created and the current number of aliases in memory. To open this monitor, execute:

```
GcGraphMorph new openInWorld
```

An example screenshot of the monitor is shown in Figure A.2. The monitor shows the total number of aliases created and the current number

of aliases in memory and it illustrates the development of these numbers over time using vertical lines. The orange vertical lines indicate the total number of aliases created and the black vertical lines indicate the number of aliases in memory. The blue vertical line shows the event of a full GC cycle.

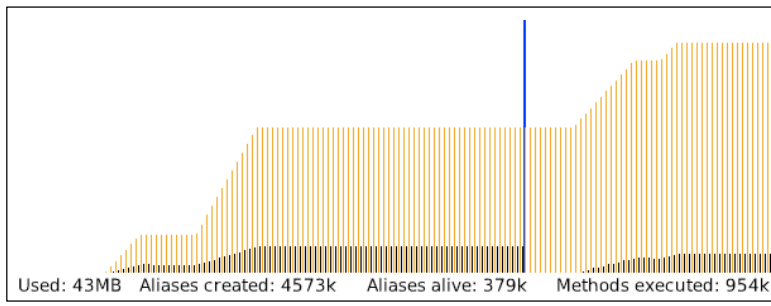

Figure A.2: Memory monitor showing aliases created and currently in memory.

A.4.3 Tuning Garbage Collector Parameters

The default parameters of the Squeak garbage collector are optimized for small object memories. The following settings change these defaults to decrease the performance overhead of the GC when lots of objects are created and the required memory is large. Depending on the memory characteristics of the application, other settings may yield better results. The meaning of these parameters are documented in the method SmalltalkImage >>vmParameterAt:.

```
Smalltalk setGCBiasToGrowGCLimit: 64*1024*1024.
Smalltalk setGCBiasToGrow: 1.
SmalltalkImage current vmParameterAt: 5 put: 10000.
SmalltalkImage current vmParameterAt: 6 put: 12000.
SmalltalkImage current vmParameterAt: 25 put: 4*1024*1024.
SmalltalkImage current vmParameterAt: 24 put: 4*1024*1024.
```

Bibliography

[AGAR 04] R. Agarwal and S. D. Stoller. *Type Inference for Parameterized Race-Free Java*. In Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04), pp 149–160, 2004. (p 3)

[ALDR 02] J. Aldrich, V. Kostadinov, and C. Chambers. *Alias Annotations for Program Understanding*. In Proceedings of the 17th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA'02), volume 37(11), pp 311–330, New York, NY, USA, November 2002. ACM. (p 3)

[ALPE 99] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. *Implementing Jalapeño in Java*. In Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'99), pp 314–324, New York, NY, USA, 1999. ACM. (p 112)

[ANTO 05] G. Antoniol and Y.-G. Guéhéneuc. *Feature Identification: a Novel Approach and a Case Study*. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM'05), pp 357–366, Los Alamitos CA, September 2005. IEEE Computer Society Press. (pp 8, 86)

[ARTZ 07] S. Artzi, A. Kiezun, D. Glasser, and M. D. Ernst. *Combined static and dynamic mutability analysis*. In Proceedings of the 22nd IEEE/ACM international conference on automated software engineering (ASE'07), pp 104–113, New York, NY, USA, 2007. ACM. (p 18)

[BALL 99] T. Ball. *The Concept of Dynamic Analysis*. In Proceedings of the European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC'99), number 1687 in LNCS, pp 216–234, Heidelberg, sep 1999. Springer Verlag. (pp 1, 86)

[BALM 01] F. Balmas. *Displaying dependence graphs: a hierarchical approach*. In Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01), p 261, Los Alamitos, CA, USA, 2001. IEEE Computer Society. (p 68)

[BANN 79] J. P. Banning. *An efficient way to find the side effects of procedure calls and the aliases of variables*. In Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL'79), pp 29–41, New York, NY, USA, 1979. ACM. (p 17)

[BECK 98] K. Beck and E. Gamma. *Test Infected: Programmers Love Writing Tests*. Java Report, vol. 3, no. 7, pp 51–56, 1998. (pp 89, 90)

[BERT 07] A. Bertolino. *Software Testing Research: Achievements, Challenges, Dreams*. In Proceedings of Future of Software Engineering (FOSE'07) at 29th International Conference on Software Engineering, pp 85–103, Washington, DC, USA, 2007. IEEE Computer Society. (pp 89, 90)

[BOND 07] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. *Tracking bad apples: reporting the origin of null and undefined value errors*. In Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications (OOPSLA'07), pp 405–422, New York, NY, USA, 2007. ACM. (p 131)

[BOOT 00] B. Boothe. *Efficient algorithms for bidirectional debugging*. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI'00), pp 299–310, New York, NY, USA, 2000. ACM. (p 135)

[BOUJ 00] A. S. Boujarwah, K. Saleh, and J. Al-Dallal. *Dynamic data flow analysis for Java programs.* Information & Software Technology, vol. 42, no. 11, pp 765–775, 2000. (p 16)

[BOYA 02] C. Boyapati, S. Khurshid, and D. Marinov. *Korat: Automated testing based on Java predicates*. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02), pp 123–133, Roma, Italy, 2002. ACM. (p 107)

[BOYA 03] C. Boyapati, B. Liskov, and L. Shrira. *Ownership types for object encapsulation*. In Principles of Programming Languages (POPL'03), pp 213–223. ACM Press, 2003. (p 3)

[BRAC 04] G. Bracha and D. Ungar. *Mirrors: design principles for meta-level facilities of object-oriented programming languages*. In Proceedings

of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices, pp 331–344, New York, NY, USA, 2004. ACM Press. (pp 113, 117)

[BRAN 98] J. Brant, B. Foote, R. Johnson, and D. Roberts. *Wrappers to the Rescue*. In Proceedings European Conference on Object Oriented Programming (ECOOP'98), volume 1445 of *LNCS*, pp 396–417. Springer-Verlag, 1998. (pp 18, 66)

[BRUN 02] E. Bruneton, R. Lenglet, and T. Coupaye. *ASM: A Code Manipulation Tool to Implement Adaptable Systems*. In Proceedings of Adaptable and Extensible Component Systems, Grenoble, France, November 2002. (p 18)

[CAIN 05] A. Cain. *Dynamic data flow analysis for object oriented programs*. PhD thesis, Swinburne University of Technology, 2005. (p 16)

[CAME 07] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. *Multiple ownership*. In Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications (OOPSLA'07), pp 441–460, New York, NY, USA, 2007. ACM. (p 31)

[CHEN 95] T. Y. Chen and C. K. Low. *Dynamic Data Flow Analysis for C++*. In Proceedings of the Second Asia Pacific Software Engineering Conference (APSEC'95), p 22, Washington, DC, USA, 1995. IEEE Computer Society. (p 16)

[CHEV 78] R. J. Chevance and T. Heidet. *Static profile and dynamic behavior of COBOL programs*. SIGPLAN Not., vol. 13, no. 4, pp 44–57, 1978. (p 5)

[CHIB 03] S. Chiba and M. Nishizawa. *An Easy-to-Use Toolkit for Efficient Java Bytecode Translators*. In In Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE'03), volume 2830 of *LNCS*, pp 364–376, 2003. (p 18)

[CLAR 01] D. G. Clarke, J. Noble, and J. M. Potter. *Simple Ownership Types for Object Containment*. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'91), LNCS, pp 53–76, London, UK, June 2001. Springer Verlag. (p 3)

[CLAR 02] D. Clarke and S. Drossopoulou. *Ownership, encapsulation and the disjointness of type and effect*. In Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'02), pp 292–310, New York, NY, USA, 2002. ACM. (p 31)

[CLAR 07] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad, editors. 3rd International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO'07). Springer, July 2007.

[DAHM 99] M. Dahm. *Byte Code Engineering*. In Proceedings of Java-Informations-Tage (JIT'99), pp 267–277, Düsseldorf, Deutschland, sep 1999. (p 18)

[DE P 94] W. De Pauw, D. Kimelman, and J. Vlissides. *Modeling Object-Oriented Program Execution*. In M. Tokoro and R. Pareschi, editors, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94), volume 821 of *LNCS*, pp 163–182, Bologna, Italy, July 1994. Springer-Verlag. (pp 5, 18, 19, 62)

[DE P 98] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. *Execution Patterns in Object-Oriented Visualization*. In Proceedings of Conference on Object-Oriented Technologies and Systems (COOTS'98), pp 219–234. USENIX, 1998. (pp 18, 67, 100)

[DE P 99] W. De Pauw and G. Sevitsky. *Visualizing Reference Patterns for Solving Memory Leaks in Java*. In R. Guerraoui, editor, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99), volume 1628 of *LNCS*, pp 116–134, Lisbon, Portugal, June 1999. Springer-Verlag. (pp 14, 100)

[DE P 00] W. De Pauw and G. Sevitsky. *Visualizing reference patterns for solving memory leaks in Java*. Concurrency: Practice and Experience, vol. 12, no. 14, pp 1431–1454, 2000. (pp 1, 3, 5, 14)

[DE P 02] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. *Visualizing the Execution of Java Programs*. In Revised Lectures on Software Visualization, International Seminar, pp 151–162, London, UK, 2002. Springer-Verlag. (pp 5, 14)

[DEME 02] S. Demeyer, S. Ducasse, and O. Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2002. (pp 8, 89)

[DEME 04] C. Demetrescu and I. Finocchi. *A portable virtual machine for program debugging and directing*. In Proceedings of the 2004 ACM symposium on Applied computing (SAC'04), pp 1524–1530, New York, NY, USA, 2004. ACM. (p 135)

[DEMS 02] B. Demsky and M. Rinard. *Role-based exploration of object-oriented programs*. In Proceedings of the 24th International Conference on Software Engineering (ICSE'02), pp 313–324, New York, NY, USA, 2002. ACM. (pp 5, 19)

[DENK 07] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. *Sub-Method Reflection*. In Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007, volume 6/9, pp 231–251. ETH, October 2007. (p 18)

[DENK 08] M. Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008. (p 66)

[DIET 07] W. Dietl and P. Müller. *Runtime Universe Type Inference*. In T. Wrigstad, editor, Proceedings of the International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO'07), Berlin, Germany, July 2007. (pp 3, 15)

[DOLA 03] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. *An Empirical Investigation of the Influence of a Type of Side Effects on Program Comprehension*. IEEE Transactions on Software Engineering, vol. 29, no. 7, pp 665–670, 2003. (p 3)

[DROS 08] S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. *A Unified Framework for Verification Techniques for Object Invariants*. In Proceedings of 22nd European Conference on Object-Oriented Programming (ECOOP'08), Lecture Notes in Computer Science, pp 412–437, July 2008. (p 31)

[DUCA 05] S. Ducasse, L. Renggli, and R. Wuyts. *SmallWiki — A Meta-Described Collaborative Content Management System*. In Proceedings ACM International Symposium on Wikis (WikiSym'05), pp 75–82, New York, NY, USA, 2005. ACM Computer Society. (p 83)

[DUCA 06] S. Ducasse, T. Gîrba, and R. Wuyts. *Object-Oriented Legacy System Trace-based Logic Testing*. In Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06), pp 35–44. IEEE Computer Society Press, 2006. (pp 20, 87, 107, 134)

[EISE 03] T. Eisenbarth, R. Koschke, and D. Simon. *Locating Features in Source Code*. IEEE Computer, vol. 29, no. 3, pp 210–224, March 2003. (pp 73, 86)

[EISE 05a] T. Eisenbarth, R. Koschke, and G. Vogel. *Static object trace extraction for programs with pointers*. Journal of Systems and Software, vol. 77, no. 3, pp 263–284, 2005. (p 17)

[EISE 05b] A. Eisenberg and K. De Volder. *Dynamic Feature Traces: Finding Features in Unfamiliar code*. In Proceedings IEEE International Conference on Software Maintenance (ICSM 2004), pp 337–346,

Los Alamitos CA, September 2005. IEEE Computer Society Press. (pp 72, 86)

[FACT 04] M. Factor, A. Schuster, and K. Shagin. *Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach*. In Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '04), pp 288–300, New York, NY, USA, 2004. ACM. (p 66)

[FELD 88] S. I. Feldman and C. B. Brown. *IGOR: a system for program debugging via reversible execution*. In Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging (PADD'88), pp 112–123, New York, NY, USA, 1988. ACM. (p 135)

[FIER 09] J. Fierz. Compass – Navigation Support for Back-in-Time Debugging. Master's thesis, University of Bern, 2009. To appear. (p 141)

[FLAN 06] C. Flanagan and S. N. Freund. *Dynamic Architecture Extraction*. In FATES/RV, volume 4262 of *Lecture Notes in Computer Science*, pp 209–224. Springer, 2006. (pp 5, 14)

[FOWL 03] M. Fowler. UML Distilled. Addison Wesley, 2003. (pp 21, 51, 90, 95)

[GODE 05] P. Godefroid, N. Klarlund, and K. Sen. *DART: directed automated random testing*. In Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation (PLDI'05), pp 213–223, New York, NY, USA, 2005. ACM. (p 107)

[GOLD 83] A. Goldberg and D. Robson. Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading, Mass., May 1983. (pp 59, 65, 120)

[GOLD 05] S. Goldsmith, R. O'Callahan, and A. Aiken. *Relational Queries over Program Traces*. In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), pp 385–402, New York, NY, USA, 2005. ACM Press. (pp 5, 19, 20, 87, 107)

[GREE 05] O. Greevy and S. Ducasse. *Correlating Features and Code Using A Compact Two-Sided Trace Analysis Approach*. In Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05), pp 314–323, Los Alamitos CA, 2005. IEEE Computer Society. (pp 18, 72, 86)

[GREE 07] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Bern, May 2007. (p 75)

[GROT 01] C. Grothoff, J. Palsberg, and J. Vitek. *Encapsulating objects with confined types*. In Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'01), pp 241–255, New York, NY, USA, 2001. ACM Press. (p 3)

[GSCH 03] T. Gschwind and J. Oberleitner. *Improving Dynamic Data Analysis with Aspect-Oriented Programming*. In Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03), p 259, Washington, DC, USA, 2003. IEEE Computer Society. (pp 5, 18, 19)

[HAMO 04] A. Hamou-Lhadj and T. Lethbridge. *A Survey of Trace Exploration Tools and Techniques*. In Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004), pp 42–55, Indianapolis IN, 2004. IBM Press. (pp 1, 8)

[HAMO 06] A. Hamou-Lhadj and T. Lethbridge. *Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System*. In Proceedings of International Conference on Program Comprehension (ICPC'06), pp 181–190, Washington, DC, USA, 2006. IEEE Computer Society. (p 18)

[HILL 00] T. Hill, J. Noble, and J. Potter. *Scalable Visualisations with Ownership Trees*. In Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00), pp 202–213, June 2000. (p 15)

[HILL 02] T. Hill, J. Noble, and J. Potter. *Scalable Visualizations of Object-Oriented Systems with Ownership Trees.* Journal of Visual Languages and Computing, vol. 13, no. 3, pp 319–339, 2002. (pp 1, 2, 3, 5, 13, 15)

[HOFE 06] C. Hofer, M. Denker, and S. Ducasse. *Design and Implementation of a Backward-In-Time Debugger*. In Proceedings of NODE'06, volume P-88 of *Lecture Notes in Informatics*, pp 17–32. Gesellschaft für Informatik (GI), September 2006. (pp 20, 110, 134)

[HOGG 91] J. Hogg. *Islands: Aliasing Protection in Object-Oriented Languages*. In Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'91), ACM SIGPLAN Notices, volume 26, pp 271–285, November 1991. (p 3)

[HOGG 92] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. *The Geneva convention on the treatment of object aliasing*. SIGPLAN OOPS Mess., vol. 3, no. 2, pp 11–16, 1992. (p 2)

[INGA 97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. *Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself*. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97), pp 318–326. ACM Press, November 1997. (pp 59, 65, 112, 120)

[JERD 97] D. Jerding and S. Rugaber. *Using Visualization for Architectural Localization and Extraction*. In I. Baxter, A. Quilici, and C. Verhoef, editors, Proceedings of 4th Working Conference on Reverse Engineering (WCRE'97), pp 56–65. IEEE Computer Society Press, 1997. (p 67)

[KANJ 99] G. K. Kanji. 100 Statistical Tests. SAGE Publications, 1999. (p 124)

[KLEY 88] M. F. Kleyn and P. C. Gingrich. *GraphTrace — Understanding Object-Oriented Systems using Concurrently Animated Views*. In Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88), volume 23, pp 191–205. ACM Press, November 1988. (p 18)

[KORE 97] B. Korel and J. Rilling. *Dynamic Program Slicing in Understanding of Program Execution*. In 5th International Workshop on Program Comprehension (WPC '97), pp 80–85, 1997. (p 1)

[KORE 98] B. Korel and J. Rilling. *Dynamic program slicing methods*. Information & Software Technology, vol. 40, no. 11-12, pp 647–659, 1998. (p 17)

[KOTH 06] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. *On Computing the Canonical Features of Software Systems*. In 13th IEEE Working Conference on Reverse Engineering (WCRE 2006), October 2006. (pp 72, 86)

[KRIN 04] J. Krinke. *Visualization of Program Dependence and Slices*. In Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), pp 168–177, Los Alamitos, CA, USA, 2004. IEEE Computer Society. (p 68)

[LANG 95] D. Lange and Y. Nakamura. *Interactive Visualization of Design Patterns can help in Framework Understanding*. In Proceedings

ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), pp 342–357, New York NY, 1995. ACM Press. (p 67)

[LENC 97] R. Lencevicius, U. Hölzle, and A. K. Singh. *Query-Based Debugging of Object-Oriented Programs*. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming (OOPSLA'97), pp 304–317, New York, NY, USA, 1997. ACM. (pp 20, 134)

[LENC 99] R. Lencevicius, U. Hölzle, and A. K. Singh. *Dynamic Query-Based Debugging*. In R. Guerraoui, editor, Proceedings of European Conference on Object-Oriented Programming (ECOOP'99), volume 1628 of *LNCS*, pp 135–160, Lisbon, Portugal, June 1999. Springer-Verlag. (pp 20, 87, 107, 134)

[LETO 86] S. Letovsky and E. Soloway. *Delocalized Plans and Program Comprehension*. IEEE Software, vol. 3, no. 3, pp 41–49, 1986. (p 3)

[LEWI 03] B. Lewis. *Debugging Backwards in Time*. In Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG'03), October 2003. (pp 1, 20, 110, 133)

[LIBL 05] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. *Scalable statistical bug isolation*. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05), pp 15–26, New York, NY, USA, 2005. ACM. (p 110)

[LIEB 98] H. Lieberman and C. Fry. *ZStep 95: A reversible, animated source code stepper*. In J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, Software Visualization — Programming as a Multimedia Experience, pp 277–292, Cambridge, MA-London, 1998. The MIT Press. (p 133)

[LIEN 07] A. Lienhard, O. Greevy, and O. Nierstrasz. *Tracking Objects to detect Feature Dependencies*. In Proceedings of International Conference on Program Comprehension (ICPC'07), pp 59–68, Washington, DC, USA, June 2007. IEEE Computer Society. (p 8)

[LIEN 08a] A. Lienhard, T. Gîrba, O. Greevy, and O. Nierstrasz. *Test Blueprints – Exposing Side Effects in Execution Traces to Support Writing Unit Tests*. In Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08), pp 83–92. IEEE Computer Society Press, 2008. (p 9)

[LIEN 08b] A. Lienhard, T. Gîrba, and O. Nierstrasz. *Practical Object-Oriented Back-in-Time Debugging*. In Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08), volume 5142 of *LNCS*, pp 592–615. Springer, 2008. ECOOP distinguished paper award. (p 9)

[LIEN 09] A. Lienhard, S. Ducasse, and T. Gîrba. *Taking an Object-Centric View on Dynamic Information with Object Flow Analysis*. Journal of Computer Languages, Systems and Structures, vol. 35, no. 1, pp 63–79, 2009. (p 8)

[LISK 86] B. Liskov and J. Guttag. Abstraction and Specification in Program Development. MIT Press/McGraw-Hill, Cambridge, Mass., USA, 1986. (p 3)

[MART 05] M. Martin, B. Livshits, and M. S. Lam. *Finding application errors and security flaws using PQL: a program query language*. In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), pp 363–385, New York, NY, USA, 2005. ACM Press. (pp 20, 134)

[MARU 03] K. Maruyama and M. Terada. *Debugging with Reverse Watchpoint*. In Proceedings of the Third International Conference on Quality Software (QSIC'03), p 116, Washington, DC, USA, 2003. IEEE Computer Society. (pp 110, 130, 131, 146)

[MEHT 02] A. Mehta and G. Heineman. *Evolving legacy systems features using regression test cases and components*. In Proceedings ACM International Workshop on Principles of Software Evolution, pp 190–193, New York NY, 2002. ACM Press. (pp 4, 86)

[MEYE 06] M. Meyer, T. Gîrba, and M. Lungu. *Mondrian: An Agile Visualization Framework*. In ACM Symposium on Software Visualization (SoftVis'06), pp 135–144, New York, NY, USA, 2006. ACM Press. (p 65)

[MITC 06] N. Mitchell. *The Runtime Structure of Object Ownership*. In Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06), volume 4067 of *Lecture Notes in Computer Science*, pp 74–98. Springer, 2006. (pp 3, 5, 15)

[NIER 05] O. Nierstrasz, S. Ducasse, and T. Gîrba. *The Story of Moose: an Agile Reengineering Environment*. In Proceedings of the European Software Engineering Conference (ESEC/FSE'05), pp 1–10, New York NY, 2005. ACM Press. Invited paper. (p 65)

[NOBL 98] J. Noble, J. Potter, and J. Vitek. *Flexible alias protection*. In E. Jul, editor, Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), volume 1445 of *LNCS*, pp 158–185, Brussels, Belgium, July 1998. Springer-Verlag. (p 3)

[OCL 06] OCL. *Object Constraint Language Specification, Version 2.0*, 2006. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf. (p 46)

[PHEN 06] S. Pheng and C. Verbrugge. *Dynamic Data Structure Analysis for Java Programs*. In Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06), pp 191–201, Washington, DC, USA, 2006. IEEE Computer Society. (pp 3, 5, 13, 14)

[POTA 04] A. Potanin, J. Noble, and R. Biddle. *Snapshot Query-Based Debugging*. In Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04), p 251, Washington, DC, USA, 2004. IEEE Computer Society. (pp 20, 134)

[POTH 07] G. Pothier, E. Tanter, and J. Piquer. *Scalable Omniscient Debugging*. Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07), vol. 42, no. 10, pp 535–552, 2007. (pp 9, 20, 110, 134)

[POTH 08] G. Pothier and É. Tanter. *Extending Omniscient Debugging to Support Aspect-Oriented Programming*. In Proceedings of the 23rd ACM Symposium on Applied Computing (SAC'08), volume 1, pp 266–270, Fortaleza, Ceará, Brazil, March 2008. (p 110)

[PRIC 93] B. A. Price, R. M. Baecker, and I. S. Small. *A Principled Taxonomy of Software Visualization*. Journal of Visual Languages and Computing, vol. 4, no. 3, pp 211–266, 1993. (pp 1, 5)

[QUAN 06] J. Quante and R. Koschke. *Dynamic Obejct Process Graphs*. In Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR 2006). IEEE Computer Society Press, 2006. (pp 31, 68)

[QUAN 07] J. Quante and R. Koschke. *Dynamic Protocol Recovery*. In Proceedings of the 14th Working Conference on Reverse Engineering (WCRE'07), pp 219–228, Washington, DC, USA, 2007. IEEE Computer Society. (p 17)

[QUAN 08] J. Quante and R. Koschke. *Dynamic object process graphs*. Journal of Systems and Software, vol. 81, no. 4, pp 481–501, 2008. (pp 17, 31)

[RAYS 06] D. Rayside, L. Mendel, and D. Jackson. *A dynamic analysis for revealing object ownership and sharing*. In Proceedings of the 2006 international workshop on Dynamic systems analysis (WODA'06), pp 57–64, New York, NY, USA, 2006. ACM. (pp 5, 15)

[RAYS 07] D. Rayside and L. Mendel. *Object ownership profiling: a technique for finding and fixing memory leaks*. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE'07), pp 194–203, New York, NY, USA, 2007. ACM. (pp 13, 15)

[REIC 07] S. Reichhart. Assessing Test Quality — TestLint. Master's thesis, University Bern, April 2007. (p 1)

[RENG 06] L. Renggli. Magritte — Meta-Described Web Application Development. Master's thesis, University of Bern, June 2006. (pp 83, 106)

[RICH 02] T. Richner. *Recovering Behavioral Design Views: a Query-Based Approach*. PhD thesis, University of Bern, May 2002. (p 18)

[RIEB 03] M. Riebisch. Towards a More Precise Definition of Feature Models, pp 64–76. BooksOnDemand Publ. Co. Norderstedt, 2003. (p 71)

[RITC 93] H. Ritch and H. M. Sneed. *Reverse Engineering Programs via Dynamic Analysis*. In Proceedings of WCRE '93, pp 192–201. IEEE, May 1993. (p 5)

[SALA 04] M. Salah and S. Mancoridis. *A Hierarchy of Dynamic Software Views: from Object-Interactions to Feature-Interacions*. In Proceedings IEEE International Conference on Software Maintenance (ICSM 2004), pp 72–81, Los Alamitos CA, 2004. IEEE Computer Society Press. (pp 8, 72, 73, 74, 79, 86, 87)

[SHAH 00] R. Shaham, E. K. Kolodner, and M. Sagiv. *On effectiveness of GC in Java*. In Proceedings of the 2nd international symposium on Memory management (ISMM'00), pp 12–17, New York, NY, USA, 2000. ACM. (p 13)

[TALP 92] J.-P. Talpin and P. Jouvelot. *Polymorphic type, region and effect inference*. Journal of Functional Programming, vol. 2, pp 245–271, 1992. (p 45)

[UTTI 06] M. Utting and B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, 2006. (p 107)

[WALK 98] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. *Visualizing Dynamic Software System Information through High-Level Models*. In Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98), pp 271–283. ACM, October 1998. (pp 5, 19, 53, 67)

[WATS 96] A. Watson and T. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Research report, National Institute of Standards and Technology, Washington, D.C., 1996. (p 107)

[WEIS 81] M. Weiser. *Program slicing*. In ICSE '81: Proceedings of the 5th international conference on Software engineering, pp 439–449, Piscataway, NJ, USA, 1981. IEEE Press. (p 17)

[WILD 92] N. Wilde and R. Huitt. *Maintenance Support for Object-Oriented Programs*. IEEE Transactions on Software Engineering, vol. SE-18, no. 12, pp 1038–1044, December 1992. (pp 2, 3)

[WILD 95] N. Wilde and M. Scully. *Software Reconnaisance: Mapping Program Features to Code*. Journal on Software Maintenance: Research and Practice, vol. 7, no. 1, pp 49–62, 1995. (pp 72, 86)

[WONG 00] E. Wong, S. Gokhale, and J. Horgan. *Quantifying the closeness between program components and features*. Journal of Systems and Software, vol. 54, no. 2, pp 87–98, 2000. (p 72)

[WRIG 94] A. K. Wright and M. Felleisen. *A syntactic approach to type soundness*. Inf. Comput., vol. 115, no. 1, pp 38–94, 1994. (p 32)

[XU 07a] H. Xu, C. J. F. Pickett, and C. Verbrugge. *Dynamic purity analysis for java programs*. In Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '07), pp 75–82, New York, NY, USA, 2007. ACM. (p 18)

[XU 07b] G. Xu, A. Rountev, Y. Tang, and F. Qin. *Efficient checkpointing of java software using context-sensitive capture and replay*. In Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE'07), pp 85–94, New York, NY, USA, 2007. ACM. (p 135)

[ZAID 05] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. *Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process*. In Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR'05),

pp 134–142, Los Alamitos CA, 2005. IEEE Computer Society Press. (pp 18, 62)

[ZAID 06] A. Zaidman, S. Demeyer, B. Adams, K. D. Schutter, G. Hoffman, and B. D. Ruyck. *Regaining Lost Knowledge through Dynamic Analysis and Aspect Orientation*. In Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'06), pp 91–102, Washington, DC, USA, 2006. IEEE Computer Society. (p 18)

[ZELL 96] A. Zeller and D. Lütkehaus. *DDD — a free graphical front-end for Unix debuggers*. SIGPLAN Not., vol. 31, no. 1, pp 22–27, 1996. (p 14)

[ZELL 05] A. Zeller. Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, October 2005. (pp 90, 110)

Curriculum Vitae

Personal Information

Name	Adrian Lienhard
Date of Birth	November 25, 1977
Place of Birth	Bern, Switzerland
Nationality	Swiss

Education

2005 – 2008	Ph.D. in Computer Science at the Software Composition Group, University of Bern, Switzerland Thesis title: *Dynamic Object Flow Analysis*
2002 – 2004	Master in Computer Science at the Software Composition Group, University of Bern, Switzerland Thesis title: *Bootstrapping Traits*
2000 – 2002	Undergraduate Degree in Computer Science at the University of Bern, Switzerland. Minors in Mathematics and Media Studies.

Complete Curriculum Vitae:
http://www.adrian-lienhard.ch/files/cv-adrianlienhard.pdf